Modular Type Classes

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

Robert Harper
Carnegie Mellon University
rwh@cs.cmu.edu

Manuel M.T. Chakravarty
Gabriele Keller
University of New South Wales
{chak,keller}@cse.unsw.edu.au

Abstract

ML modules and Haskell type classes have proven to be highly effective tools for program structuring. Modules emphasize explicit configuration of program components and the use of data abstraction. Type classes emphasize implicit program construction and *ad hoc* polymorphism. In this paper, we show how the implicitly-typed style of type class programming may be supported within the framework of an explicitly-typed module language by viewing type classes as a particular mode of use of modules. This view offers a harmonious integration of modules and type classes, where type class features, such as class hierarchies and associated types, arise naturally as uses of existing module-language constructs, such as module hierarchies and type components. In addition, programmers have explicit control over which type class instances are available for use by type inference in a given scope. We formalize our approach as a Harper-Stone-style elaboration relation, and provide a sound type inference algorithm as a guide to implementation.

1. Introduction

The ML module system [17] and the Haskell type class system [25, 19] have proved, through more than 15 years of practical experience and theoretical analysis, to be effective linguistic tools for structuring programs. Each provides the means of *specifying* the functionality of program components, *abstracting* programs over such specifications, and *instantiating* programs with specific realizations of the specifications on which they depend. In ML such specifications are called *signatures*, abstraction is achieved through *functors,* and instantiation is achieved by *functor application* to *structures* that implement these signatures. In Haskell such specifications are called *type classes*, abstraction is achieved through *constrained polymorphism,* and instantiation is achieved through *polymorphic instantiation* with *instances* of type classes. There is a clear correspondence between the highlighted concepts (see [26]), and consequently modules and type classes are sometimes regarded as opposing approaches to language design. We show that there is no opposition. Rather, type classes and modules are complementary aspects of a comprehensive framework of modularity.

Perhaps the most significant difference is the mode of use of the two concepts. The Haskell type class system is primarily intended to support *ad hoc* polymorphism in the context of a parametrically polymorphic language. It emphasizes the *implicit* inference of class constraints and *automatic* construction of instances during overload resolution, which makes it convenient to use in many common cases, but does not facilitate more general purposes of modular programming. Moreover, the emphasis on automatic generation of instances imposes inherent limitations on expressiveness—most importantly, there can be at most one instance of a type class at any particular type.

In contrast, the ML module system is designed to support the structuring of programs by forming hierarchies of components and imposing abstraction boundaries—both *client-side* abstraction, via functors, and *implementor-side* abstraction, via signature ascription (aka *sealing*). The module system emphasizes *explicit* manipulation of modules in the program, which makes it more flexible and general than the type class mechanism. Modules may be ascribed multiple signatures that reveal varying amounts of type information, signatures may be implemented by many modules, and neither modules nor signatures are restricted to have the rigid form that Haskell's instances and classes have. On the other hand, ML lacks support for *implicit* module generation and *ad hoc* polymorphism, features which experience with Haskell has shown to be convenient and desirable.

There have been many proposals to increase the expressiveness of the original type class system as proposed by Wadler and Blott [25], including constructor classes [15], functional dependencies [12], named instances [16], and associated types [2, 1]. These may all be seen as adding functionality to the Haskell class system that mirrors aspects of the ML module system, while retaining the implicit style of usage of type classes. However, these (and other) extensions tend to complicate the type class system without alleviating the underlying need for a more expressive module system.

In fact, there are ways in which the Haskell type class mechanism impedes modularity. To support implicit instance generation while ensuring coherence of inference, Haskell insists that instances of type classes be drawn from a global set of instance declarations; in particular, instances are implicitly exported and imported, which puts their availability beyond programmer control. This can be quite inconvenient—for many type classes there is more than one useful instance of the class at a particular type, and the appropriate choice of instance depends on the context in which an overloaded operator is used. Hence, the Haskell Prelude must provide many functions in two versions: one using type classes and the other an explicit function argument—e.g., `sort` and `sortBy`.

In this paper we take a different tack. Rather than bolster the expressiveness of type classes, we instead propose that a more sensible approach to combining the benefits of type classes and modules is to *start* with modules as the fundamental concept, and *then* recover type classes as a particular mode of use of modularity. We retain the advantages of a fully expressive *explicit* module system, while also offering the conveniences of *implicit* type class programming, particularly the integration of *ad hoc* and parametric polymorphism. Moreover, the proposed design provides a clean separation between the *definition* of instances and their *availability for use* during type inference. This offers localization of instance scoping, enhanced readability, and the potential for instances to be compiled separately from their uses. The result is a harmonious integration of modules and type classes that provides the best features of both approaches in a single, consistent framework. The elegance of our approach stems from the observation that type class features, such as class hierarchies and associated types, arise naturally as uses of existing module-language constructs, such as module hierarchies and type components.

In summary, this paper makes the following contributions:

- We present a smooth integration of type classes and modules that provides a foundation for future work on incorporating type classes into ML and a proper module system into Haskell. We give an intuition of the integration of type classes into ML in Section 2.
- We highlight some interesting design issues that arose while developing the interpretation of type classes in terms of modules (Section 3).
- We specify the semantics of an extended module language that supports type classes. We formalize its elaboration (in the style of Harper and Stone [9]) into an explicitly-typed module type system. We also generalize Damas and Milner's Algorithm W [3] to an inference algorithm for modular type classes that we have proved sound with respect to the elaboration semantics (Section 4).

Our elaboration translation demonstrates that modules can serve as *evidence* in the sense of Jones [14]. Compared to the customary use of dictionary records as evidence, modules offer a cleaner way of handling extensions to the basic type class mechanism such as associated types. In addition, for the application to type classes, the use of modules as evidence makes clear that the construction of evidence respects the phase distinction [8], *i.e.,* it is based solely on compile-time information, not run-time information. We conclude in Section 5 with further discussion of related work.

2. Modular Type Classes: An Overview

In this section we summarize our approach to representing the main mechanisms of a Haskell-style type class system within the context of an ML-style module system. For readability, we employ ML-like syntax for our examples, although the formal design we describe later is syntactically more austere and leaves a number of (largely superficial) aspects of an actual ML extension to future work.

2.1 Classes are signatures, instances are modules

A type class in Haskell is essentially an interface describing a set of operations whose types mention a distinguished abstract type variable known as the *class parameter.* It is natural therefore to represent a class in the module setting as a signature (*i.e.,* an interface) with a distinguished type component (representing the class parameter). In particular, we insist that the distinguished type component be named "`t`". It may be followed by any number of other type, value, or substructure components. We call such a signature a *class signature*, specifically an *atomic* class signature (in contrast to the *composite* ones that we describe below in Section 2.3.) For example, the class of equality types is represented by the atomic class signature `EQ`, defined as follows:

```
signature EQ = sig
  type t
  val eq : t * t -> bool
end
```

Note that class signatures like `EQ` are just ordinary ML signatures of a certain specified form.

Correspondingly, an instance of a type class is represented by a module. A monomorphic instance of a type class is represented by a structure, and a polymorphic instance is represented by a functor. For example, we can encode an `int` instance of the equality class as a structure whose signature is `EQ where type t = int`:

```
structure EqInt = struct
  type t = int
  val eq = Int.eq
end
```

As in Haskell, the instance for a compound type $t(t_1, \ldots, t_n)$ is composed from instances of its component types, $t_1, \ldots, t_n$, by a functor, `Eq`$_t$, associated with its outermost type constructor, t. For example, here is an instance of equality for product types $t_1 * t_2$:

```
functor EqProd (X : EQ, Y : EQ) = struct
  type t = X.t * Y.t
  fun eq ((x1,y1), (x2, y2)) =
    X.eq(x1,x2) andalso Y.eq(y1,y2)
end
```

There is an evident correspondence with Haskell instance declarations, but rather than use Horn clause logic programs to specify closure conditions, we instead use functional programs (in the form of functors).

From the `EqInt` and `EqProd` modules we can construct an instance, say, of signature `EQ where type t=int*int`:

```
structure EqII = EqProd(EqInt,EqInt)
```

Of course, one of the main reasons for using type classes in the first place is so that we don't have to write this functor application manually—it corresponds to the process known as dictionary construction in Haskell and can be performed automatically, behind the scenes, during type inference. In particular, such automatic functor application may occur in the elaboration of expressions that appear to be values, such as when a variable undergoes polymorphic instantiation (see below). Consequently, it is important that the application of an instance functor does not engender any computational effects, such as I/O or non-termination. We therefore require that instance functors be *total* in the sense that their bodies satisfy something akin to ML's value restriction. This restriction appears necessary in order to ensure predictable program behavior.

2.2 Separating the definition of an instance from its use

In Haskell, an instance becomes immediately available for use by the type inference engine as soon as it is declared. As a consequence, due to the implicit global importing and exporting of instances, there can only ever be a single instance of a class at a certain type in one program. This is often a nuisance and leads to awkward workarounds. Proposals such as named instances [16] have attempted to alleviate this problem, but have not been generally accepted.

In contrast, our reconstruction of type classes in terms of modules provides a natural solution to this dilemma. Specifically, we require that an instance module only become available for use by the inference engine after it has been nominated for this purpose explicitly by a `using` declaration. This separates the *definition* of an instance from its *adoption* as a *canonical instance*, thus facilitating modular decomposition and constraining inference to make use only of a clearly specified set of instances. For example, the declaration

```
using EqInt, EqProd in mod
```

nominates the two instance modules defined earlier as available for canonical instance generation during elaboration of the module *`mod`*. The typing rule for `using` demands that `EqInt` and `EqProd` not overlap with any instances that have already been adopted as canonical. (A precise definition of overlapping instances is given in Section 3.2.)

In both our language and Haskell, canonical instance generation is implicitly invoked whenever overloading is resolved. In our language, we additionally provide a mechanism `canon(`*`sig`*`)` by which the programmer can explicitly request the canonical instance module implementing the class signature *sig*.[1] At whatever point within *mod* instance generation occurs, it will employ only those instances that have been adopted as canonical in that scope.

2.3 Class hierarchies via module hierarchies

In Haskell, one can extend a class A with additional operations to form a class B, at which point A is called a *superclass* of B. Class hierarchies arise in the module setting naturally from module hierarchies. This is easiest to illustrate by example.

Suppose we want to define a class called ORD, which extends the EQ class with a lt operation. We can do this by first defining an atomic class LT that only supports lt, and then defining ORD as a *composite* of EQ and LT:

```
signature ORD = sig
  structure E : EQ
  structure L : LT
  sharing type E.t = L.t
end
```

The sharing specification makes explicit that ORD is providing two different interpretations of the *same* type, as an equality type and as an ordered type. ORD is an example of what we call a *composite class signature*, *i.e.*, a signature consisting of a collection of atomic signatures bound to submodules whose names are arbitrary.

Instances of composite class signatures are not written by the programmer directly, but rather are composed automatically by the inference engine from the instances for their atomic signature parts. For example, if we want to write instances of ORD for int and the * type constructor, what we do instead is to write instances of LT:

```
structure LtInt = struct
  type t = int
  val lt = Int.lt
end
functor LtProd (X : ORD, Y : LT) = struct
  type t = X.E.t * Y.t
  fun lt ((x1,y1), (x2,y2)) =
    X.L.lt(x1,x2) orelse
      (X.E.eq(x1,x2) andalso Y.L.lt(y1,y2))
end
```

Note that LtProd requires its first argument to be an instance of ORD, not LT. This is because the implementation of lt in the body of the functor depends on having both equality and ordering on the type X.E.t so that it can implement a lexicographic ordering on X.E.t * Y.t. For Y.t, only the lt operation is needed.

Now, let us assume these instances are made canonical (via the using declaration) in a certain scope. Then, during typechecking, if the inference engine demands a canonical module of signature ORD where type E.t = int * int, it will be computed to be

```
struct
  structure E = EqProd(EqInt,EqInt)
  structure L = LtProd(struct
                         structure E = EqInt
                         structure L = LtInt
                       end,
                       LtInt)
end
```

The fundamental reason that we do not allow instances for ORD to be adopted directly is that we wish to prevent the instances for ORD from having any overlap with existing instances that may have been adopted for EQ. If one were to define an instance for ORD where type E.t = int directly, one would implicitly provide an instance for EQ where type t = int through its E substructure; and if one tried to adopt such an ORD instance as canonical, it would overlap with any existing canonical instance of EQ where type t = int.

Under our approach, this sort of overlap is avoided. Moreover, the code one writes is ultimately very similar to the code one would write in Haskell (except that it is expressed entirely in terms of existing ML constructs). In particular, the instance declaration for ORD at int in Haskell is only permitted to provide a definition for the new operations (namely, lt) that are present in ORD but not in EQ. In other words, an instance declaration for ORD in Haskell is precisely what we would call an instance of LT.

2.4 Constrained polymorphism via functors

Under the Harper-Stone interpretation of Standard ML (hereafter, HS) [9], polymorphic functions in the *external* (source) language are elaborated into *functors* in an *internal* module type system. Specifically, a polymorphic value is viewed as a functor that takes a module consisting only of type components (representing the polymorphic type variables) as its argument and returns a module consisting of a single value component as its result.

The HS semantics supports the concept of *equality polymorphism* found in Standard ML by simply extending the class of signatures over which polymorphic functions may be abstracted to include the EQ signature defined above. For example, in the internal module type system of HS, the *ad hoc* polymorphic equality function is represented by the functor

```
functor eq (X:EQ) :> ⟦X.t * X.t -> bool⟧ = [X.eq]
```

where the brackets notation describes a module with a single value component. Polymorphic instantiation at a type τ consists of computing a canonical instance of EQ where type t = τ, as described above, applying the functor eq to it, and extracting the value component of the resulting module.

The present proposal is essentially a generalization of the HS treatment of equality polymorphism to arbitrary type classes. A functor that abstracts over a module representing an instance of a type class is reminiscent of the notion of a *qualified type* [11], except that we make use of the familiar concept of a functor from the ML module system, rather than introduce a new mechanism solely to support *ad hoc* polymorphism.

Of course, the programmer need not write the eq functor manually. Our external language provides an overload mechanism, and the elaborator will generate the above functor automatically when the programmer writes

```
val eq = overload eq from EQ
```

Note that there is no need to bind the polymorphic function returned by the overload mechanism to the name eq; it can be called anything. In practice, it may be useful to be able to overload all the components of a class signature at once by writing overload SIG as syntactic sugar for a sequence of overload's for the individual components of the signature.

The following are some examples of elaboration in the presence of the overloaded eq function:

```
   using EqInt, EqProd in ...eq((2,3),(4,5))...
⇝ ...Val(eq(EqProd(EqInt,EqInt))) ((2,3),(4,5))...

   fun refl y = eq(y,y)
⇝ functor refl (X : EQ) :> ⟦X.t -> bool⟧
     = [fn y => Val(eq(X)) (y,y)]
```

[1] This feature is particularly useful in conjunction with our support for associated types; see Section 2.5.

(Note: the `Val` operator seen here is the mechanism in our internal module type system by which a value of type τ is extracted from a module of signature $[\![\tau]\!]$.)

Our language also allows for the possibility that the programmer may wish to work with explicitly polymorphic functions in addition to implicit overloaded ones. In particular, by writing

```
functor Refl = explicit (refl :
  (X : EQ) -> sig val it : X.t -> bool end)
```

we convert the polymorphic function `refl` into an explicit functor `Refl`. The programmer can then apply it to an arbitrary module argument of signature `EQ` and project out the `it` component of the result. The reason we require a signature annotation on the `explicit` construct is that the implicitly-typed `refl` may be declaratively ascribed many different signatures. Whenever `refl` is used, type inference will compute the appropriate instance arguments for it regardless of the particular signature it has been ascribed. However, since it is the *programmer* who applies `Refl`, she needs to know exactly what shape `Refl`'s module argument is expected to have.

We also provide an `implicit` construct to coerce explicit functors into implicit ones. (See Section 4.2 for details.)

2.5 Associated types arise naturally

The experience with type classes in Haskell quickly led to the desire for type classes with more than one class parameter. However, these multi-parameter type classes are not generally very useful unless dependencies between the parameters can be expressed. This led in turn to the proposal of functional dependencies [12] and more recently associated types [2, 1] for Haskell.

An associated type is a type component provided by a class that is not the distinguished type component (class parameter). The associated types of a class do not play a role in determining the canonical instance of a class at a certain type—that is solely determined by the identity of the distinguished type.

Modular type classes immediately support associated types as additional type components of a class signature. An illustrative example is provided by a class of collection types:

```
signature COLLECTS = sig
  type t
  type elem
  val empty  : t
  val insert : elem * t -> t
  val member : elem * t -> bool
  val toList : t -> elem list
end
```

The distinguished type `t` represents the collection type and the associated type component `elem` represents the type of elements. An instance for lists, where the elements are required to support equality for the membership test, would be defined as follows:

```
functor CollectsList (X : EQ) = struct
  type t = X.t list
  type elem = X.t
  val empty = []
  fun insert (x, L) = x::L
  fun member (x, []) = raise NotInCollection
    | member (x, y::L) = X.eq (x,y) orelse
                         member (x,L)
  fun toList L = L
end
```

When using classes with associated types, it is common to need to place some constraints on the identities of the associated types. For example, suppose we write the following:

```
val toList = overload toList from COLLECTS
fun sumColl C = sum (toList C)
```

The `sumColl` function does not care what type of collection `C` is, so long as its element type is `int`. Correspondingly, the elaborator will assign `sumColl` the polymorphic type (*i.e.,* functor signature)

```
(X : COLLECTS where type elem = int) -> ⟦X.t -> int⟧
```

Note that the constraint on the type `X.elem` is expressed completely naturally using ML's existing `where type` mechanism, which is just syntactic sugar for the transparent realization of an abstract type component in a signature. In contrast, the extension to handle associated type synonyms in Haskell [1] requires an additional mechanism called *equality constraints* in order to handle functions like `sumColl`.

As Chakravarty *et al.* [1] have demonstrated, it is useful in certain circumstances to be able to compute (statically) the identity of an associated type `assoc` in the canonical instance of a type class `SIG` at a given type τ. This is achieved in our setting via the `canon(`*sig*`)` construct, which we introduced above as a way of explicitly computing a canonical instance. In particular, we can write

```
canon(SIG where type t = τ).assoc
```

which constructs the canonical instance of `SIG` at τ and then projects the `assoc` type from it.[2] In the associated type extension to Haskell, one would instead write `assoc(`τ`)`.

While the ML syntax here is clearly less compact, there is a good reason for it. Specifically, the Haskell syntax only makes sense because Haskell ties each associated type name in the program to a single class (in this case, `assoc` would be tied to `SIG`). In contrast, in our setting, it is fine for several different class signatures to have an associated type component called `assoc`.

3. Design Considerations

In this section we examine some of the more subtle points in the design of modular type classes and explain our approach to handling them.

3.1 Coherence in the presence of scoped instances

The `using` mechanism described in the introduction separates the definition of instance functors from their adoption as canonical instances. It also raises questions of coherence stemming from the nondeterministic nature of polymorphic type inference. Suppose `EqInt1` and `EqInt2` are two observably distinct instances of `EQ where type t = int`. Consider the following code:

```
structure A = using EqInt1 in
  struct ...fun f x = eq(x,x)... end
structure B = using EqInt2 in
  struct ...val y = A.f(3)... end
```

The type inference algorithm is free to resolve the meaning of this program in two incompatible ways. On the one hand, it may choose to treat `A.f` as polymorphic over the class `EQ`; in this case, the application `A.f(3)` demands an instance of `EQ where type t=int`, which can only be resolved by `EqInt2`. On the other hand, type inference is free to assign the type `int -> bool` to `A.f` at the point where `f` is defined, in which case the demand for an instance of `EQ` can only be met by `EqInt1`. *These are both valid typings, but they lead to observably different behavior.*

[2] Note that, due to the principle of *phase separation* in the ML module system [8], the identity of the `assoc` type here can be determined purely statically, and elaboration does not actually need to construct the dynamic parts of `canon(SIG where type t = `τ`)`.

An unattractive solution is to insist on a specific algorithm for type inference that arbitrarily chooses one resolution over another, but this sacrifices the elegant, declarative nature of a Hindley-Milner-style type system and, worse, imposes a specific resolution policy that may not be desired in practice. Instead, we prefer to take a different approach, which is to put the decision under programmer control, permitting either outcome at her discretion. We could achieve this by insisting that the scope of a `using` declaration be given an explicit signature, so that in the above example the programmer would have to specify whether `A.f` is to be polymorphic or monomorphic. However, this approach is awkward for nested `using` declarations, forcing repeated specifications of the same information.

Instead we propose that the `using` declaration be confined to an *outer* (or *top-level*) layer that consists only of module declarations, whose signatures are typically specified in any case. All core-level terms appear in the *inner* layer, where type inference proceeds without restriction, but no `using` clauses are allowed. Thus, the set of permissible instances is fixed in any inner context, but may vary across outer contexts. At the boundary of the two layers, a type or signature annotation is required. This ensures that the scope of a `using` declaration is explicitly typed without effecting duplication of annotations. The programmer who wishes to ignore type classes simply confines herself to the inner level, with no restrictions; only the use of type classes demands attention be paid to the distinction.

3.2 Overlapping instances

To ensure coherence of type inference, the set of available instances in any context must be *non-overlapping*. Roughly speaking, this means that there should only be one way to compute the canonical instance of any given class at any given type. There is considerable leeway, though, in determining the precise definition of overlap, and indeed this remains a subject of debate in the Haskell community. For the purposes of this paper we follow the guidelines used in Haskell 98. In particular, we insist that there be one instance per type constructor, so that instance resolution proceeds by a simple inductive analysis of the structure of the instance type, composing instance functors to obtain the desired result.

However, in the modular approach suggested here, there is an additional complication. Just as a module may satisfy several different signatures, so a single module may qualify as an instance of several different type classes. For example, the module

```
struct type t = int; fun f(x:t) = x end
```

may be seen as an instance of the class

```
sig type t; val f : t -> t end
```

and also of the class

```
sig type t; val f : t -> int end.
```

Thus, to check if two instances `A` and `B` (with the same `t` component) are non-overlapping, we need to ensure that the set of *all* classes to which `A` could belong is disjoint from the set of *all* classes to which `B` could also belong.

A simple, but practical, criterion to ensure this is to define two instances to be non-overlapping iff either (1) they differ on their distinguished `t` component, so that no overlap is possible, or (2) in the case that they have the same `t` component, that they be *structurally dissimilar*, which we define to mean that their components do not all have the same names and appear in the same order. While other, more refined definitions are possible, we opt here for simplicity until evidence of the need for a more permissive criterion is available.

3.3 Unconstrained type components in class signatures

In order to support ordinary ML-style polymorphism, we need a way to include unconstrained type components in a class signature. We could use the class signature `sig type t end` for this purpose. However, since our policy is that the only canonical instances of atomic class signatures are those that have been explicitly adopted as canonical by a `using` declaration, this would amount to treating `sig type t end` as a special case.

We choose instead to allow composite class signatures to contain arbitrary unconstrained type components, so long as they are named something other than `t`. For example, under our approach, the divergent function

```
fun f x = f x
```

can be assigned the polymorphic type

$$\texttt{(X : sig type a; type b end) -> } [\![\texttt{X.a -> X.b}]\!]$$

(The choice of the particular names `a` and `b` here is arbitrary.)

In our formal system, we refer to the union of the `t` components and the unconstrained components of a class signature S as the *parameters* of S.

3.4 Multi-parameter and constructor classes

Two extensions to Wadler & Blott's [25] type class system that have received considerable attention are *multi-parameter type classes* and *constructor classes*. We have chosen not to cover these extensions in this paper. Concerning multi-parameter classes, most uses of them require functional dependencies [12], which when rewritten to use associated types (which we support), turn into single-parameter classes. Hence, we expect the need for multi-parameter classes to be greatly diminished in our case.

As for constructor classes, we see no fundamental problems in supporting them in an extension of our framework since type components of ML modules may have higher kind. However, we view them as an orthogonal extension, and thus have opted to omit them in the interest of a clearer and more compact presentation.

4. Formal System

In this section we formalize our language design in the style of Harper and Stone [9]. This consists of an elaboration translation of programs from an external source language to an internal module type system. The elaboration translation is syntax-directed, but it is also nondeterministic with respect to polymorphic generalization and instantiation. To show how this language may be implemented, we define a type inference algorithm, which we have proven sound.

Elaboration translations are the standard method of giving meaning to programs involving type classes, although in the context of Haskell they are often called *evidence translations* [14]. The crucial difference between our account and previous treatments is that we elaborate into a module-aware internal language.

4.1 Internal language type system

Figure 1 shows the syntax and module typing rules for our internal language (IL). (The remainder of the type system is given in Figure 5 of the appendix.) The IL we use here is a simplified variant of the type system for modules defined in Dreyer's thesis [4], which in turn is based on the higher-order module calculus of Dreyer, Crary and Harper [6].

The core and module levels of our language are tightly coupled. The core, or term, fragment is relatively standard, so we concentrate primarily on the module fragment. The basic module constructs are the type (constructor) module $[\mathrm{C}]$ and the term module $[e]$, which contain exactly one component each. These modules are given the signatures $[\![\mathrm{K}]\!]$ and $[\![\tau]\!]$, respectively, assuming that C has kind K,

$$
\begin{array}{lrl}
\text{Kinds} & \mathrm{K},\mathrm{L} ::= & \mathbf{T} \mid \mathfrak{S}(\tau) \mid \{\overline{\ell{\triangleright}\alpha{:}\mathrm{K}}\} \mid \Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2 \\
\text{Transparent Kinds} & \mathbb{K},\mathbb{L} ::= & \mathfrak{S}(\tau) \mid \{\overline{\ell{\triangleright}\alpha{:}\mathbb{K}}\} \mid \Pi\alpha{:}\mathrm{K}_1.\mathbb{K}_2 \\
\text{Constructors} & \mathrm{C},\tau ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \{\overline{\ell{\triangleright}\alpha{=}\mathrm{C}}\} \mid \mathrm{C}.\ell \mid \lambda\alpha{:}\mathrm{K}.\mathrm{C} \mid \mathrm{C}_1(\mathrm{C}_2) \\
\text{Terms} & e,f ::= & x \mid \lambda x{:}\tau.e \mid e_1(e_2) \mid \mathsf{Val}(\mathrm{M}) \mid \mathsf{let}\,\mathrm{X}{=}\mathrm{M}\,\mathsf{in}\,e \\
\text{Valuable Terms} & v,u ::= & x \mid \lambda x{:}\tau.e \mid \mathsf{Val}(\mathrm{V}) \mid \mathsf{let}\,\mathrm{X}{=}\mathrm{V}\,\mathsf{in}\,v \\
\text{Signatures} & \mathrm{S},\mathrm{R} ::= & \llbracket\mathrm{K}\rrbracket \mid \llbracket\tau\rrbracket \mid \{\overline{\ell{\triangleright}\mathrm{X}{:}\mathrm{S}}\} \mid \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \mid \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \\
\text{Transparent Signatures} & \mathbb{S},\mathbb{R} ::= & \llbracket\mathbb{K}\rrbracket \mid \llbracket\tau\rrbracket \mid \{\overline{\ell{\triangleright}\mathrm{X}{:}\mathbb{S}}\} \mid \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \mid \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \\
\text{Modules} & \mathrm{M},\mathrm{N},\mathrm{F} ::= & \mathrm{X} \mid [\mathrm{C}] \mid [e] \mid \{\overline{\ell{\triangleright}\mathrm{X}{=}\mathrm{M}}\} \mid \mathbb{M}.\ell \mid \\
 & & \lambda(\mathrm{X}{:}\mathrm{S}_1){:>}\mathrm{S}_2.\mathrm{M} \mid \mathrm{F}(\mathrm{M}) \mid \Lambda\mathrm{X}{:}\mathrm{S}_1.\mathrm{V} \mid \mathrm{F}\langle\mathrm{M}\rangle \mid \\
 & & \mathrm{M} :> \mathrm{S} \mid \mathsf{let}\,\mathrm{X}{=}\mathrm{M}_1\,\mathsf{in}\,\mathrm{M}_2 :> \mathrm{S} \\
\text{Projectible Modules} & \mathbb{M},\mathbb{N},\mathbb{F} ::= & \mathrm{X} \mid [\mathrm{C}] \mid [e] \mid \{\overline{\ell{\triangleright}\mathrm{X}{=}\mathbb{M}}\} \mid \mathbb{M}.\ell \mid \\
 & & \lambda(\mathrm{X}{:}\mathrm{S}_1){:>}\mathrm{S}_2.\mathrm{M} \mid \Lambda\mathrm{X}{:}\mathrm{S}_1.\mathbb{V} \mid \mathbb{F}\langle\mathbb{M}\rangle \\
\text{Valuable Modules} & \mathrm{V},\mathrm{U} ::= & \mathrm{X} \mid [\mathrm{C}] \mid [v] \mid \{\overline{\ell{\triangleright}\mathrm{X}{=}\mathrm{V}}\} \mid \mathbb{V}.\ell \mid \\
 & & \lambda(\mathrm{X}{:}\mathrm{S}_1){:>}\mathrm{S}_2.\mathrm{M} \mid \Lambda\mathrm{X}{:}\mathrm{S}_1.\mathrm{V} \mid \mathrm{V}_1\langle\mathrm{V}_2\rangle \mid \\
 & & \mathrm{V} :> \mathbb{S} \mid \mathsf{let}\,\mathrm{X}{=}\mathrm{V}_1\,\mathsf{in}\,\mathrm{V}_2 :> \mathbb{S} \\
\text{Contexts} & \Gamma ::= & \emptyset \mid \Gamma,\alpha{:}\mathrm{K} \mid \Gamma,x{:}\tau \mid \Gamma,\mathrm{X}{:}\mathrm{S}
\end{array}
$$

Well-formed Modules: $\Gamma \vdash \mathrm{M} : \mathrm{S}$

$$
\frac{\mathrm{X}{:}\mathrm{S} \in \Gamma}{\Gamma \vdash \mathrm{X} : \mathrm{S}}
\qquad
\frac{\Gamma \vdash \mathrm{C} : \mathrm{K}}{\Gamma \vdash [\mathrm{C}] : \llbracket\mathrm{K}\rrbracket}
\qquad
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] : \llbracket\tau\rrbracket}
\qquad
\frac{}{\Gamma \vdash \{\} : \{\}}
$$

$$
\frac{\Gamma \vdash \mathrm{M}_1 : \mathrm{S}_1 \quad \Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \{\overline{\ell{\triangleright}\mathrm{X}{=}\mathrm{M}}\} : \{\overline{\ell{\triangleright}\mathrm{X}{:}\mathrm{S}}\}}{\Gamma \vdash \{\ell_1{\triangleright}\mathrm{X}_1{=}\mathrm{M}_1,\overline{\ell{\triangleright}\mathrm{X}{=}\mathrm{M}}\} : \{\ell_1{\triangleright}\mathrm{X}_1{:}\mathrm{S}_1,\overline{\ell{\triangleright}\mathrm{X}{:}\mathrm{S}}\}}
\qquad
\frac{\Gamma \vdash \mathbb{M} : \{\ell_1{\triangleright}\mathrm{X}_1{:}\mathrm{S}_1,\ldots,\ell_n{\triangleright}\mathrm{X}_n{:}\mathrm{S}_n\} \quad i \in 1..n}{\Gamma \vdash \mathbb{M}.\ell_i : \mathrm{S}_i[\mathbb{M}.\ell_j/\mathrm{X}_j]_{j=1}^{i-1}}
$$

$$
\frac{\Gamma \vdash \mathrm{S}_1\ \mathrm{sig} \quad \Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M} : \mathrm{S}_2}{\Gamma \vdash \lambda(\mathrm{X}{:}\mathrm{S}_1){:>}\mathrm{S}_2.\mathrm{M} : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2}
\quad
\frac{\Gamma \vdash \mathrm{F} : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \quad \Gamma \vdash \mathbb{M} : \mathrm{S}_1}{\Gamma \vdash \mathrm{F}(\mathbb{M}) : \mathrm{S}_2[\mathbb{M}/\mathrm{X}]}
\quad
\frac{\Gamma \vdash \mathrm{S}_1\ \mathrm{sig} \quad \Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{V} : \mathbb{S}_2}{\Gamma \vdash \Lambda\mathrm{X}{:}\mathrm{S}_1.\mathrm{V} : \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2}
\quad
\frac{\Gamma \vdash \mathrm{F} : \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \quad \Gamma \vdash \mathbb{M} : \mathrm{S}_1}{\Gamma \vdash \mathrm{F}\langle\mathbb{M}\rangle : \mathbb{S}_2[\mathbb{M}/\mathrm{X}]}
$$

$$
\frac{\Gamma \vdash \mathrm{M}_1 : \mathrm{S}_1 \quad \Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M}_2 : \mathrm{S} \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathrm{S})}{\Gamma \vdash \mathsf{let}\,\mathrm{X}{=}\mathrm{M}_1\,\mathsf{in}\,\mathrm{M}_2 :> \mathrm{S} : \mathrm{S}}
\qquad
\frac{\Gamma \vdash \mathrm{M} : \mathrm{S}}{\Gamma \vdash \mathrm{M} :> \mathrm{S} : \mathrm{S}}
\qquad
\frac{\Gamma \vdash \mathrm{M} : \mathrm{S}' \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S}}{\Gamma \vdash \mathrm{M} : \mathrm{S}}
\qquad
\frac{\Gamma \vdash \mathbb{M} : \mathrm{S}}{\Gamma \vdash \mathbb{M} : \mathfrak{S}_{\mathrm{S}}(\mathbb{M})}
$$

Figure 1. Internal Language Syntax and Module Typing Rules

and e has type τ. Compound modules, or *structures*, are dependent records of the form

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

Here, $\ell_1,\ldots,\ell_n$ are the pairwise distinct *labels*, or *external names*, by which the record components are accessed, and $\mathrm{X}_1,\ldots,\mathrm{X}_n$ are the *variables*, or *internal names*, by which subsequent components can refer to previous ones (see [7] for more on this essential distinction). The type, or *signature*, of such a record has the form

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

which we abbreviate by writing $\{\overline{\ell{\triangleright}\mathrm{X}{:}\mathrm{S}}\}$. (Throughout the paper, we will use $\overline{E}$ as a shorthand for $E_1,\ldots,E_n$, for various syntactic constructs E.) The variables mediate the dependencies among the signatures of the successive components of the record.

Module signatures are *translucent* in that they may reveal the definitions of some (including all or none) of their type components. We model this through the use of *singleton kinds* [23, 24]. Briefly, a constructor of kind $\mathfrak{S}(\tau)$ is definitionally equivalent to τ. In particular, if this is the kind of a variable, then the variable may be thought of as having τ as its definition. As a limiting case, a signature is *transparent* iff the kinds of all of its type constructor components are singletons—the definitions of all components are thereby revealed.

A *functor* is simply a function at the level of modules, *albeit* one with a dependent type expressing the flow of type information from argument to result. Here, as in the HS semantics, we make essential use of both *total* and *partial* functors. Total functors, written with a Λ, must have *valuable* (pure and terminating) bodies; partial functors, written with a λ, impose no restrictions. Consequently, the application of a total functor to a valuable argument, written $\mathrm{V}_1\langle\mathrm{V}_2\rangle$, is itself valuable, whereas application of a partial functor, written $\mathrm{F}(\mathrm{M})$, is not. In addition to requiring that the bodies of total functors be valuable, we also require that they have fully transparent result signatures. This arises from the interpretation of data abstraction as a computational effect, as described by Dreyer, *et al.* [6].

The module language provides mechanisms for let-binding a module and sealing a module with a signature ascription. The typing rules for these constructs (and a number of others) are only useful in conjunction with the signature subsumption rule, which allows a module to be coerced to a less transparent signature using the signature subtyping judgment. The definition of signature subtyping (see Figure 5 in the appendix) is, however, fairly rigid. In particular, it does not allow dropping or reordering of components from structure signatures, and it coincides with signature equivalence at functor signatures. A more flexible notion of signature subtyping is provided by the *coercive signature matching* judgment in the elaboration relation given below.

Finally, it is essential to review the constructs for extracting the type and value components of modules, which provide the interface between the core and module levels of the language. The term $\mathsf{Val}(\mathrm{M})$ extracts the term component from the module, M, of signature $\llbracket\tau\rrbracket$. One might expect an analogous operation at the type level, but instead we employ a meta-operation $\mathsf{Fst}(\mathrm{M})$ that *computes* the type component of the module M of signature $\llbracket\mathrm{K}\rrbracket$. When M is a variable, X, the projection $\mathsf{Fst}(\mathrm{X})$ is defined to be an associated type variable X^{c} of kind K. Otherwise, $\mathsf{Fst}(\mathrm{M})$ is

Kinds	$knd ::=$	$\mathbf{T} \mid \mathfrak{S}(typ) \mid \mathbf{T}^n \to \mathbf{T} \mid \Pi\overline{\alpha}.\mathfrak{S}(typ)$
Transparent Kinds	$tknd ::=$	$\mathfrak{S}(typ) \mid \Pi\overline{\alpha}.\mathfrak{S}(typ)$
Type Constructors	$con, typ ::=$	$\alpha \mid typ_1 \to typ_2 \mid \mathrm{P} \mid \mathsf{canon}(sig).\ell s \mid \lambda\overline{\alpha}.typ \mid con(\overline{typ})$
Terms	$exp ::=$	$x \mid \lambda x.exp \mid exp_1(exp_2) \mid \mathrm{P} \mid \mathsf{let}\,\mathrm{X}{=}mod\,\mathsf{in}\,exp \mid exp : typ$
Signatures	$sig ::=$	$\llbracket knd \rrbracket \mid \llbracket typ \rrbracket \mid \{\overline{\ell \triangleright \mathrm{X}{:}sig}\} \mid \Pi\mathrm{X}{:}sig_1.sig_2 \mid \forall\mathrm{X}{:}sig_1.tsig_2$
Transparent Signatures	$tsig ::=$	$\llbracket tknd \rrbracket \mid \llbracket typ \rrbracket \mid \{\overline{\ell \triangleright \mathrm{X}{:}tsig}\} \mid \Pi\mathrm{X}{:}sig_1.sig_2 \mid \forall\mathrm{X}{:}sig_1.tsig_2$
Modules	$mod ::=$	$\mathrm{P} \mid [con] \mid [exp] \mid \{\overline{\ell \triangleright \mathrm{X}{=}mod}\} \mid \lambda\mathrm{X}{:}sig_1.mod \mid \mathrm{P}_1(\mathrm{P}_2) \mid$ $\Lambda\mathrm{X}{:}sig_1.mod \mid \mathrm{P}_1\langle\mathrm{P}_2\rangle \mid \mathsf{let}\,\mathrm{X}{=}mod_1\,\mathsf{in}\,mod_2 \mid mod \mathrel{:>} sig \mid$ $\mathsf{canon}(sig) \mid \mathsf{overload}\ \ell s\ \mathsf{from}\ sig \mid \mathsf{implicit}(\mathrm{P}) \mid \mathsf{explicit}(\mathrm{P}:\mathrm{S})$
Top-Level Modules	$top ::=$	$\mathrm{P} \mid [con] \mid \{\overline{\ell \triangleright \mathrm{X}{=}top}\} \mid \lambda\mathrm{X}{:}sig_1.top \mid \mathrm{P}_1(\mathrm{P}_2) \mid$ $\Lambda\mathrm{X}{:}sig_1.top \mid \mathrm{P}_1\langle\mathrm{P}_2\rangle \mid \mathsf{let}\,\mathrm{X}{=}top_1\,\mathsf{in}\,top_2 \mid mod \mathrel{:>} sig \mid$ $\mathsf{canon}(sig) \mid \mathsf{overload}\ \ell s\ \mathsf{from}\ sig \mid \mathsf{implicit}(\mathrm{P}) \mid \mathsf{explicit}(\mathrm{P}:\mathrm{S}) \mid$ $\mathsf{using}\ \mathrm{P}\ \mathsf{in}\ top$
Sequences of Label Projections	$\ell s ::=$	$\ell \mid \ell.\ell s$
Constructor Paths	$p ::=$	$\alpha \mid \alpha.\ell s$
Module Paths	$\mathrm{P} ::=$	$\mathrm{X} \mid \mathrm{X}.\ell s$
Instance Sets	$\Theta ::=$	$\emptyset \mid \Theta, \mathrm{P}$

Figure 2. External Language Syntax

defined inductively on the structure of M. In keeping with the interpretation of type abstraction as an effect [6], not all modules permit extraction of their type components. Those that do—that is, those for which $\mathsf{Fst}(\mathbb{M})$ is defined—are said to be *projectible*. Since variables are deemed projectible, any module that is substituted for a variable must also be projectible. (See [4, 6] for details of projectibility.)

4.2 External language

Figure 2 shows the syntax of our external language (EL), which is elaborated into the internal language described above.

The EL type language is similar to that of ML. In particular, EL type constructors are restricted to be of kind $\mathbf{T}$ or $\mathbf{T}^n \to \mathbf{T}$ (short for $\Pi\alpha{:}\{1{:}\mathbf{T}, \ldots, n{:}\mathbf{T}\}.\mathbf{T}$). In addition, types may be projected (through one or more label projections) only from modules that are variables (X) or that have the form $\mathsf{canon}(sig)$. Note that the EL does not have polymorphic types, because we interpret polymorphism using functors.

The EL term language is essentially an implicitly-typed version of the IL term language. The sealing construct, $exp : typ$, is used to annotate a term with a specific type. Term variables (x) are monomorphic, whereas paths (P) which are compositions of projections from modules, may be polymorphic. This is consistent with our treatment of polymorphic functions as functors.

The syntax of EL signatures is similar to that of IL signatures, except that we include a special form of functor signature for the representation of polymorphism. The main difference compared to a general total functor is that the argument signature must represent a type class, and the result signature must be that of an atomic value module. A signature is deemed a type class if it is a collection of unconstrained type components and atomic instance components (whose first component is `t`), in which the unconstrained and `t` components are all abstract (possibly subject to sharing constraints). We omit explicit `where` or `sharing` constructs here, since these can be simulated using type definitions in signatures.

Polymorphic generalization takes place when a term is injected into the module language (using the atomic module $[exp]$), and polymorphic instantiation takes place when a module path P appears in a core-language term. This resonates with the idea (due to Harper and Stone) of interpreting the classical distinction between polytypes and monotypes as a distinction between the module and core levels of the language. It also divorces so-called *let-polymorphism* from dependence on a let construct. Specifically, the traditional polymorphic let construct $\mathsf{let}\,x{=}exp_1\,\mathsf{in}\,exp_2$ is encodable in our language as $\mathsf{let}\,\mathrm{X}{=}[exp_1]\,\mathsf{in}\,\{x \mapsto \mathrm{X}\}exp_2$.

The EL module constructs $\mathsf{canon}(sig)$ and $\mathsf{overload}\ \ell s\ \mathsf{from}\ sig$ are described in Section 2.4; these are not present in the IL, but are instead elaborated into uses of the IL module constructs. Similarly, the constructs $\mathsf{implicit}(\mathrm{P})$ and $\mathsf{explicit}(\mathrm{P} : sig)$ convert between implicit and explicit forms of polymorphic values. The explicit form of a polymorphic value of signature $\forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket$ is a functor of signature $\forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket\tau\rrbracket\}$, which can be constructed and applied manually by the programmer.

Finally, as described in Section 3.1, there is a distinction between inner-level modules (mod) and top-level modules (top). The syntax for these levels is nearly identical, except that core-level terms only appear in mod's, and using declarations only appear in top's. The only point at which mod's may enter the syntax of top's is in the construct $mod \mathrel{:>} sig$, where they are annotated with the signature sig.

4.3 Elaboration

A selection of judgments and rules for elaboration of the EL into the IL are given in Figure 3. The overall structure is derived from the Harper-Stone semantics of ML, much of which carries over essentially unchanged. We concentrate here only on those aspects related to type classes. Please see the appendix for a complete specification of the IL, EL, and elaboration.

The main elaboration judgments take as input a context comprised of an IL typing context Γ and a *canonical instance set* Θ. The latter consists of a set of paths to structures and functors that have been adopted for use in canonical instance generation.

The elaboration of type constructors is straightforward. Rule 1 converts module paths of signature $\llbracket\mathrm{K}\rrbracket$ to constructors of kind K by applying the meta-operation Fst described in Section 4.1. Rule 2 computes projections from a $\mathsf{canon}(sig)$ module similarly.

The elaboration of signatures is straightforward as well. In the case of total functor signatures, we must do case analysis to check whether the result signature is of the form $\{\ldots\}$ or $\llbracket\tau\rrbracket$. In the latter case, the signature represents a constrained polymorphic type, so Rule 4 ensures that the argument signature is a valid class signature. This is achieved by the class elaboration judgment, but we defer explanation of this important judgment until we discuss polymorphic generalization below.

Type Constructors: $\Theta;\Gamma \vdash \mathit{con} \leadsto \mathrm{C} : \mathrm{K}$

$$\frac{\Gamma \vdash \mathrm{P} : \llbracket \mathrm{K} \rrbracket}{\Theta;\Gamma \vdash \mathrm{P} \leadsto \mathsf{Fst}(\mathrm{P}) : \mathrm{K}}\ (1) \qquad \frac{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \leadsto \mathbb{V} : \mathbb{S} \quad \Gamma \vdash \mathbb{V}.\ell s : \llbracket \mathrm{K} \rrbracket}{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}).\ell s \leadsto \mathsf{Fst}(\mathbb{V}).\ell s : \mathrm{K}}\ (2)$$

Signatures: $\Theta;\Gamma \vdash \mathit{sig} \leadsto \mathrm{S}$

$$\frac{\Theta;\Gamma \vdash \mathit{sig}_1 \leadsto \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{tsig}_2 \leadsto \mathbb{S}_2 \quad \mathbb{S}_2 = \{\ldots\}}{\Theta;\Gamma \vdash \forall \mathrm{X}{:}\mathit{sig}_1.\mathit{tsig}_2 \leadsto \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2}\ (3) \qquad \frac{\begin{array}{c}\Theta;\Gamma \vdash \mathit{sig} \leadsto \mathrm{S} \quad \Gamma \vdash_{\mathrm{class}} \mathrm{X} : \mathrm{S} \leadsto \Theta' \\ \Theta,\Theta';\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathit{typ} \leadsto \tau : \mathbf{T}\end{array}}{\Theta;\Gamma \vdash \forall \mathrm{X}{:}\mathit{sig}.\llbracket \mathit{typ} \rrbracket \leadsto \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket}\ (4)$$

Terms: $\Theta;\Gamma \vdash \mathit{exp} \leadsto e : \tau$

$$\frac{x{:}\tau \in \Gamma}{\Theta;\Gamma \vdash x \leadsto x : \tau}\ (5) \qquad \frac{\Gamma \vdash \tau_1 : \mathbf{T} \quad \Theta;\Gamma,x{:}\tau_1 \vdash \mathit{exp} \leadsto e : \tau_2}{\Theta;\Gamma \vdash \lambda x.\mathit{exp} \leadsto \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ (6) \qquad \frac{\Theta;\Gamma \vdash \mathit{exp}_1 \leadsto e_1 : \tau_2 \to \tau \quad \Theta;\Gamma \vdash \mathit{exp}_2 \leadsto e_2 : \tau_2}{\Theta;\Gamma \vdash \mathit{exp}_1(\mathit{exp}_2) \leadsto e_1(e_2) : \tau}\ (7)$$

$$\frac{\Gamma \vdash \mathrm{P} : \llbracket \tau \rrbracket}{\Theta;\Gamma \vdash \mathrm{P} \leadsto \mathsf{Val}(\mathrm{P}) : \tau}\ (8) \qquad \frac{\Gamma \vdash \mathrm{P} : \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket \quad \Gamma \vdash \mathbb{S} \leq \mathrm{S} \quad \Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}}{\Theta;\Gamma \vdash \mathrm{P} \leadsto \mathsf{Val}(\mathrm{P}\langle \mathbb{V} \rangle) : \tau[\mathbb{V}/\mathrm{X}]}\ (9) \qquad \frac{\Theta;\Gamma \vdash \mathit{typ} \leadsto \tau : \mathbf{T} \quad \Theta;\Gamma \vdash \mathit{exp} \leadsto e : \tau}{\Theta;\Gamma \vdash \mathit{exp} : \mathit{typ} \leadsto e : \tau}\ (10)$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod} \leadsto \mathrm{M} : \mathrm{S} \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathit{exp} \leadsto e : \tau \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\tau)}{\Theta;\Gamma \vdash \mathsf{let}\,\mathrm{X}{=}\mathit{mod}\ \mathsf{in}\ \mathit{exp} \leadsto \mathsf{let}\,\mathrm{X}{=}\mathrm{M}\ \mathsf{in}\ e : \tau}\ (11) \qquad \frac{\Theta;\Gamma \vdash \mathit{exp} \leadsto e : \tau' \quad \Gamma \vdash \tau' \equiv \tau : \mathbf{T}}{\Theta;\Gamma \vdash \mathit{exp} \leadsto e : \tau}\ (12)$$

Modules: $\Theta;\Gamma \vdash \mathit{mod} \leadsto \mathrm{M} : \mathrm{S}$

$$\frac{\Theta;\Gamma \vdash \mathit{exp} \leadsto e : \tau}{\Theta;\Gamma \vdash [\mathit{exp}] \leadsto [e] : \llbracket \tau \rrbracket}\ (13) \qquad \frac{\mathrm{X} \notin \mathrm{FV}(\mathit{exp}) \quad \Gamma \vdash_{\mathrm{class}} \mathrm{X} : \mathrm{S} \leadsto \Theta' \quad \Theta,\Theta';\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathit{exp} \leadsto v : \tau}{\Theta;\Gamma \vdash [\mathit{exp}] \leadsto \Lambda \mathrm{X}{:}\mathrm{S}.[v] : \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket}\ (14)$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \leadsto \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod} \leadsto \mathrm{V} : \mathbb{S}_2 \quad \mathbb{S}_2 = \{\ldots\}}{\Theta;\Gamma \vdash \Lambda \mathrm{X}{:}\mathit{sig}.\mathit{mod} \leadsto \Lambda \mathrm{X}{:}\mathrm{S}_1.\mathrm{V} : \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2}\ (15) \qquad \frac{\begin{array}{c}\Gamma \vdash \mathrm{P}_1 : \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S} \quad \mathbb{S} = \{\ldots\} \\ \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \quad \Theta;\Gamma \vdash \mathrm{P}_2 \preceq \mathrm{S}_1 \leadsto \mathbb{V} : \mathbb{S}_1\end{array}}{\Theta;\Gamma \vdash \mathrm{P}_1\langle \mathrm{P}_2 \rangle \leadsto \mathrm{P}_1\langle \mathbb{V} \rangle : \mathbb{S}[\mathbb{V}/\mathrm{X}]}\ (16)$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \leadsto \mathrm{S} \quad \Gamma \vdash \mathrm{S}\ \mathsf{concrete} \quad \Gamma \vdash \mathbb{S} \leq \mathrm{S} \quad \Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}}{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \leadsto \mathbb{V} : \mathbb{S}}\ (17) \qquad \frac{\Theta;\Gamma \vdash \mathit{sig} \leadsto \mathrm{S} \quad \Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathrm{X}.\ell s : \llbracket \tau \rrbracket \quad \Gamma \vdash \mathrm{S}\ \mathsf{class}}{\Theta;\Gamma \vdash \mathsf{overload}\ \ell s\ \mathsf{from}\ \mathit{sig} \leadsto \Lambda \mathrm{X}{:}\mathrm{S}.(\mathrm{X}.\ell s) : \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket}\ (18)$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \leadsto \mathrm{S}_1 \quad \Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \mathrm{X}_1.\ell s : \forall \mathrm{X}_2{:}\mathrm{S}_2.\llbracket \tau \rrbracket \quad \mathrm{S} = \{1{\triangleright}\mathrm{X}_1{:}\mathrm{S}_1, 2{\triangleright}\mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\ \mathsf{class}}{\Theta;\Gamma \vdash \mathsf{overload}\ \ell s\ \mathsf{from}\ \mathit{sig} \leadsto \Lambda \mathrm{X}{:}\mathrm{S}.(\mathrm{X}.1.\ell s\langle \mathrm{X}.2 \rangle) : \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau[\mathrm{X}.i/\mathrm{X}_i] \rrbracket}\ (19)$$

Coercive Signature Matching: $\Theta;\Gamma \vdash \mathrm{P} \preceq \mathrm{S} \leadsto \mathbb{V} : \mathbb{S}$

$$\frac{\Gamma \vdash \mathrm{P} : \llbracket \mathbb{K} \rrbracket \quad \Gamma \vdash \mathbb{K} \leq \mathrm{K}}{\Theta;\Gamma \vdash \mathrm{P} \preceq \llbracket \mathrm{K} \rrbracket \leadsto \mathrm{P} : \llbracket \mathbb{K} \rrbracket}\ (20) \qquad \frac{\mathbb{S}\ \text{is of the form}\ \llbracket \tau \rrbracket\ \text{or}\ \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket \quad \Theta;\Gamma \vdash [\mathrm{P}] \leadsto \mathbb{V} : \mathbb{S}}{\Theta;\Gamma \vdash \mathrm{P} \preceq \mathbb{S} \leadsto \mathbb{V} : \mathbb{S}}\ (21)$$

Top-Level Modules: $\Theta;\Gamma \vdash \mathit{top} \leadsto \mathrm{M} : \mathrm{S}$

$$\frac{\Theta;\Gamma \vdash \mathrm{P}\ \mathsf{usable} \quad \Theta,\mathrm{P};\Gamma \vdash \mathit{top} \leadsto \mathrm{M} : \mathrm{S}}{\Theta;\Gamma \vdash \mathsf{using}\ \mathrm{P}\ \mathsf{in}\ \mathit{top} \leadsto \mathrm{M} : \mathrm{S}}\ (22)$$

Canonical Modules: $\Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}$

$$\frac{\Gamma \vdash \tau : \mathbf{T}}{\Theta;\Gamma \vdash_{\mathrm{can}} [\tau] : \llbracket \mathfrak{S}(\tau) \rrbracket}\ (23) \qquad \frac{\Gamma,\alpha{:}\mathbf{T}^n \vdash \tau : \mathbf{T}}{\Theta;\Gamma \vdash_{\mathrm{can}} [\lambda\alpha{:}\mathbf{T}^n.\tau] : \llbracket \Pi\alpha{:}\mathbf{T}^n.\mathfrak{S}(\tau) \rrbracket}\ (24) \qquad \frac{\forall i \in 1..n :\ \Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V}_i : \mathbb{S}_i \quad \mathtt{t} \notin \{\ell_1,\ldots,\ell_n\}}{\Theta;\Gamma \vdash_{\mathrm{can}} \{\ell_1{=}\mathbb{V}_1,\ldots,\ell_n{=}\mathbb{V}_n\} : \{\ell_1{:}\mathbb{S}_1,\ldots,\ell_n{:}\mathbb{S}_n\}}\ (25)$$

$$\frac{\mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} : \mathbb{S} \quad \mathbb{S} = \{\mathtt{t}{:}\llbracket \mathfrak{S}(\tau) \rrbracket,\ldots\}}{\Theta;\Gamma \vdash_{\mathrm{can}} \mathrm{P} : \mathbb{S}}\ (26) \qquad \frac{\begin{array}{c}\mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} : \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \\ \Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}_1 \quad \Gamma \vdash \mathbb{S}_1 \leq \mathrm{S}_1\end{array}}{\Theta;\Gamma \vdash_{\mathrm{can}} \mathrm{P}\langle \mathbb{V} \rangle : \mathbb{S}_2[\mathbb{V}/\mathrm{X}]}\ (27) \qquad \frac{\Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}' \quad \Gamma \vdash \mathbb{S}' \equiv \mathbb{S}}{\Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}}\ (28)$$

Class Elaboration: $\Gamma \vdash \mathrm{S}\ \mathsf{class} \quad \Gamma \vdash_{\mathrm{class}} \mathrm{X} : \mathrm{S} \leadsto \Theta$

$$\frac{\Gamma \vdash_{\mathrm{class}} \mathrm{X} : \mathrm{S} \leadsto \Theta}{\Gamma \vdash \mathrm{S}\ \mathsf{class}}\ (29) \qquad \frac{\Gamma \vdash \mathrm{S}\ \ \mathrm{sig} \quad \mathrm{S} \rightrightarrows \exists\overline{\alpha}.\mathbb{S} \quad \Gamma,\overline{\alpha} \vdash \mathbb{S} \downarrow \mathbb{S}' \quad \mathsf{params}(\mathbb{S}') \subseteq \{\overline{\alpha}\} \quad \Gamma,\overline{\alpha},\mathrm{X}{:}\mathbb{S} \vdash \mathsf{paths}(\mathrm{X} : \mathbb{S}) \downarrow \Theta}{\Gamma \vdash_{\mathrm{class}} \mathrm{X} : \mathrm{S} \leadsto \Theta}\ (30)$$

Figure 3. Key Elaboration Rules

Concerning term elaboration, the first three and the last three rules shown in Figure 3 are standard. The rules for λ-abstractions and applications are nondeterministic in the choice of argument type, as is typical for a declarative specification of elaboration (see [18, 9]).

Rule 8 governs the projection of a value from a monomorphic path P of signature $[\![\tau]\!]$. If P has the polymorphic signature $\forall \mathrm{X}{:}\mathrm{S}.[\![\tau]\!]$, then it must be instantiated before its value component can be accessed. This is governed by Rule 9, which specifies that instantiation consists of finding the *canonical* instance module of the class signature S to which P will be applied. Since the parameters of S are abstract, the choice of which instance module is nondeterministic. Correspondingly, what the second premise does is to pick a transparent subsignature $\mathbb{S}$ of S that realizes these parameters with some choices $\tau_1, \ldots, \tau_n$. Then, the third premise finds the canonical module $\mathbb{V}$ of signature $\mathbb{S}$ using the *canonical module* judgment $\Theta; \Gamma \vdash_{\overline{\mathrm{can}}} \mathbb{V} : \mathbb{S}$, described below. Note that all of this is done in terms of module and signature judgments, without ever explicitly mentioning the instantiating types $\tau_1, \ldots, \tau_n$!

The atomic term module $[exp]$ can be elaborated monomorphically (Rule 13) or polymorphically (Rule 14). The polymorphic option is only available if exp elaborates to a valuable term v, per the usual value restriction. One can view Rule 14 as "guessing" a polymorphic type $\forall \mathrm{X}{:}\mathrm{S}.[\![\tau]\!]$ to assign to exp. Suppose that S is an atomic class signature like `EQ`. In order to see whether exp can be elaborated with this type, we add the class constraint $\mathrm{X}{:}\mathrm{S}$ to the context and make it a canonical instance of the signature S `where type t =` X`.t` (by adding X to Θ) before typechecking exp. The last step is critical: if X is not added to Θ, then the canonical module judgment will have no way of knowing that X is the canonical module of signature S `where type t =` X`.t` at polymorphic instantiation time.

However, in the case that S is a composite class, the elaborator does not permit X to be added directly to the instance set Θ. To simplify the formalization of other judgments, we require all the instance structures in Θ to have atomic signature. Thus, in general we need a way of parsing the class constraint $\mathrm{X}{:}\mathrm{S}$ in order to produce a *set* of paths Θ' (all of which are rooted at X) that represent the atomic instance modules contained within X. This class parsing is achieved via the *class elaboration* judgment $\Gamma \vdash_{\overline{\mathrm{class}}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta'$ used in the second premise of Rule 14. (The judgment also checks that S is a valid class signature.)

For example, if S were the composite class `ORD` from Section 2.3, then Θ' would be the set $\{\mathrm{X}.\mathtt{E}, \mathrm{X}.\mathtt{L}\}$. Note that the instances in Θ' are guaranteed not to overlap with any instances in the input instance set Θ because the instances in Θ' all concern abstract type components of the freshly chosen variable X.

Total functors elaborate successfully so long as their bodies are valuable transparent structures (Rule 15). Rule 16 elaborates total functor applications $\mathrm{P}_1\langle \mathrm{P}_2 \rangle$ by matching the argument P_2 against P_1's argument signature S_1 via the *coercive signature matching* judgment, $\Theta; \Gamma \vdash \mathrm{P}_2 \preceq \mathrm{S}_1 \rightsquigarrow \mathbb{V} : \mathbb{S}_1$. This judgment (taken directly from the HS semantics) takes as input a path and a target signature, and returns a module derived from the input path that matches the target signature. Although the transparent signature $\mathbb{S}_1$ describing $\mathbb{V}$ is not used in this particular rule, it is guaranteed to be a subtype of the target signature S_1.

Rule 17 elaborates $\mathsf{canon}(sig)$ by computing the (unique) canonical module of signature sig. This operation is only possible if the parameters of sig are *concrete, i.e.,* they are all transparently equivalent to types that are well-formed in Γ. This simple check is performed by an auxiliary *concreteness* judgment, $\Gamma \vdash \mathrm{S}\ \mathsf{concrete}$, whose definition is given in Figure 13 in the appendix. Note that the concreteness of sig does not imply that it is *fully* transparent, only that its parameters are transparent. In particular, the *associated* type components in sig may be abstract.

Rules 18 and 19 translate the $\mathsf{overload}\ \ell s\ \mathsf{from}\ sig$ mechanism essentially as prescribed in Section 2.4. The latter rule is useful for overloading a value component of sig that already has a polymorphic type. The class constraint on that value component is joined with sig itself to form a composite class constraint.

Rules 20 and 21 illustrate the base cases of the coercive signature matching judgment. To coerce to an atomic kind signature $[\![\mathrm{K}]\!]$, the input path P must be an atomic type module whose component is of kind K. To coerce to a (potentially polymorphic) type signature $\mathbb{S}$, it must be the case that P has a more general polymorphic type than $\mathbb{S}$. This subsumption check is captured very concisely by checking whether the η-expansion of P (with respect to constrained type abstraction) can be assigned the signature $\mathbb{S}$.

Rule 22 elaborates $\mathsf{using}\ \mathrm{P}\ \mathsf{in}\ top$ by first checking whether P is *usable* and then adding P to the canonical instance set Θ during the elaboration of top. Usability is determined by the *usable instance* judgment $\Theta; \Gamma \vdash \mathrm{P}\ \mathsf{usable}$ (defined in Figure 13 of the appendix). This judgment specifies formally what we described informally in Section 3.2, and thus guarantees that P will not overlap with any of the instances in Θ.

Rules 23–28 define the canonical module judgment. In short, a composite instance module is canonical if all its atomic instance components are canonical; an atomic instance module is canonical if it is either a canonical instance structure (from the set Θ) or the result of applying a canonical instance functor from Θ to a canonical argument. Canonical modules may also contain arbitrary unconstrained type components (named something other than `t`).

Finally, Rule 30 defines the class elaboration judgment, written $\Gamma \vdash_{\overline{\mathrm{class}}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta$, which checks whether S is a class signature and then parses the class constraint $\mathrm{X}{:}\mathrm{S}$ into a set of paths to the atomic instances in X (see the discussion of polymorphic generalization above). The premises of Rule 30 refer to several auxiliary judgments and meta-operations defined in Figure 12 in the appendix.

The first premise checks that S is well-formed. The second creates a sequence of type variables $\overline{\alpha}$ corresponding to the abstract type components of S. It also returns $\mathbb{S}$, a transparent version of S with the abstract type specifications replaced by references to $\overline{\alpha}$. For example, $[\![\mathbf{T}]\!] \rightrightarrows \exists \alpha.[\![\boldsymbol{\mathfrak{S}}(\alpha)]\!]$. The third premise normalizes $\mathbb{S}$ to $\mathbb{S}'$ (using the normalization algorithm of Stone and Harper [24]). If S is indeed a valid class signature, then this step should render all the *parameters* of the class (*i.e.,* the `t` components together with the unconstrained components) transparently equal to one of the $\overline{\alpha}$. This is precisely what the fourth premise checks.

The last premise of Rule 30 computes the paths to the atomic instance modules within X and reduces them to a normal form. Reduction to normal form ensures that none of the paths overlap with each other. For instance, S might be a composite class containing two substructures of class `EQ` with a shared `t` component. Such an S is a perfectly legitimate class, but the Θ' returned by elaboration of S can only contain the path to one of the two substructures.

4.4 Type inference algorithm

The elaboration relation presented above is nondeterministic, and hence is not directly implementable without backtracking. In this section we present a deterministic type inference algorithm in the style of Algorithm $\mathcal{W}$ [3]. In particular, we thread through the inference rules a substitution δ whose domain consists of *unification variables*, denoted by bold $\boldsymbol{\alpha}$.

In addition, polymorphic instantiation in the presence of type classes generates *constraints*, which we denote Σ. Constraints are sets of $\mathrm{X}{:}\mathbb{S}$ bindings, in which the X's do not appear free in the $\mathbb{S}$'s. Each $\mathrm{X}{:}\mathbb{S}$ represents a demand generated by the algorithm for

$$
\begin{array}{rcl}
\mathsf{partition}(\overline{\boldsymbol{\alpha}}; \Sigma) & \overset{\mathrm{def}}{=} & (\Sigma_1; \Sigma_2), \\
 & & \text{where } \Sigma_2 = \{\mathrm{X}{:}\mathbb{S} \mid \mathrm{X}{:}\mathbb{S} \in \Sigma \wedge \exists \boldsymbol{\alpha} \in \overline{\boldsymbol{\alpha}}.\, \mathbb{S} = \{\mathtt{t}{:}\boldsymbol{\mathfrak{S}}(\boldsymbol{\alpha}), \ldots\}\} \text{ and } \Sigma_1 = \Sigma - \Sigma_2 \\
\mathsf{makesig}(\overline{\boldsymbol{\alpha}}; \Sigma) & \overset{\mathrm{def}}{=} & \{\mathsf{tyvars}{\triangleright}\mathrm{Y}{:}\{1{:}\llbracket \mathbf{T} \rrbracket, \ldots, m{:}\llbracket \mathbf{T} \rrbracket\}, \mathsf{consts}{:}\{1{:}\mathbb{S}'_1, \ldots, n{:}\mathbb{S}'_n\}\}, \\
 & & \text{where } \overline{\boldsymbol{\alpha}} = \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m \text{ and } \Sigma = \mathrm{X}_1{:}\mathbb{S}_1, \ldots, \mathrm{X}_n{:}\mathbb{S}_n \text{ and } \mathbb{S}'_i = \mathbb{S}_i[\mathrm{Y}.j/\boldsymbol{\alpha}_j]_{j=1}^{n} \\
\mathsf{genvars}(\Gamma; \Sigma; \tau) & \overset{\mathrm{def}}{=} & \overline{\boldsymbol{\alpha}}_2, \\
 & & \text{where } \overline{\boldsymbol{\alpha}}_2 \text{ is the greatest set such that } \overline{\boldsymbol{\alpha}}_1 \cup \overline{\boldsymbol{\alpha}}_2 = \mathrm{UV}(\Sigma, \tau) \text{ and } \overline{\boldsymbol{\alpha}}_1 \cap \overline{\boldsymbol{\alpha}}_2 = \emptyset \\
 & & \text{and } \overline{\boldsymbol{\alpha}}_2 \cap \mathrm{UV}(\Gamma, \Sigma_1) = \emptyset, \text{ where } \mathsf{partition}(\overline{\boldsymbol{\alpha}}_2; \Sigma) = (\Sigma_1; \Sigma_2)
\end{array}
$$

$\boxed{\textbf{Type Unification: } \Gamma \vdash \tau_1 \equiv \tau_2 \Rightarrow \delta \quad \Gamma \vdash \tau_1 = \tau_2 \Rightarrow \delta}$

$$
\frac{\Gamma \vdash \tau_1 \downarrow \tau'_1 \quad \Gamma \vdash \tau_2 \downarrow \tau'_2 \quad \Gamma \vdash \tau'_1 = \tau'_2 \Rightarrow \delta}{\Gamma \vdash \tau_1 \equiv \tau_2 \Rightarrow \delta} \;(31) \qquad \frac{\boldsymbol{\alpha} \notin \mathrm{FV}(\tau) \quad \vdash \Gamma[\tau/\boldsymbol{\alpha}]\ \mathsf{ok}}{\Gamma \vdash \boldsymbol{\alpha} = \tau \Rightarrow \{\boldsymbol{\alpha} \mapsto \tau\}} \;(32)
$$

$\boxed{\textbf{Terms: } \Theta; \Gamma \vdash \mathit{exp} \Rightarrow e : \tau/(\Sigma; \delta)}$

$$
\frac{x{:}\tau \in \Gamma}{\Theta; \Gamma \vdash x \Rightarrow x : \tau/(\emptyset; \mathbf{id})} \;(33) \qquad \frac{\Theta; \Gamma, x{:}\boldsymbol{\alpha} \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma; \delta)}{\Theta; \Gamma \vdash \lambda x.\mathit{exp} \Rightarrow \lambda x{:}\delta\boldsymbol{\alpha}.e : \delta\boldsymbol{\alpha} \rightarrow \tau/(\Sigma; \delta|_{\Gamma})} \;(34)
$$

$$
\frac{\Theta; \Gamma \vdash \mathit{exp}_1 \Rightarrow\!\downarrow e_1 : \tau_1/(\Sigma_1; \delta_1) \quad \Theta; \delta_1\Gamma \vdash \mathit{exp}_2 \Rightarrow\!\downarrow e_2 : \tau_2/(\Sigma_2; \delta_2) \quad \delta_2\delta_1\Gamma \vdash \delta_2\tau_1 \equiv (\tau_2 \rightarrow \boldsymbol{\alpha}) \Rightarrow \delta_3}{\Theta; \Gamma \vdash \mathit{exp}_1(\mathit{exp}_2) \Rightarrow \delta_3\delta_2 e_1(\delta_3 e_2) : \delta_3\boldsymbol{\alpha}/(\delta_3\delta_2\Sigma_1, \delta_3\Sigma_2; \delta_3\delta_2\delta_1|_{\Gamma})} \;(35)
$$

$$
\frac{\Gamma \vdash \mathrm{P} :\downarrow \llbracket \tau \rrbracket}{\Theta; \Gamma \vdash \mathrm{P} \Rightarrow \mathsf{Val}(\mathrm{P}) : \tau/(\emptyset; \mathbf{id})} \;(36) \qquad \frac{\Gamma \vdash \mathrm{P} :\downarrow \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket \quad \mathrm{S} \rightrightarrows \exists \overline{\boldsymbol{\alpha}}.\mathbb{S} \quad \Gamma, \mathrm{X}{:}\mathbb{S} \vdash \tau \downarrow \tau'}{\Theta; \Gamma \vdash \mathrm{P} \Rightarrow \mathsf{Val}(\mathrm{P}\langle \mathrm{X} \rangle) : \tau'/(\mathrm{X}{:}\mathbb{S}; \mathbf{id})} \;(37)
$$

$\boxed{\textbf{Terms With Constraint and Type Normalization: } \Theta; \Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma; \delta)}$

$$
\frac{\Theta; \Gamma \vdash \mathit{exp} \Rightarrow e : \tau'/(\Sigma_1; \delta_1) \quad \Theta; \delta_1\Gamma \vdash \Sigma_1 \downarrow (\Sigma_2; \sigma; \delta_2) \quad \delta_2\delta_1\Gamma \vdash \delta_2\tau' \downarrow \tau}{\Theta; \Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow \sigma\delta_2 e : \tau/(\Sigma_2; \delta_2\delta_1|_{\Gamma})} \;(38)
$$

$\boxed{\textbf{Modules: } \Theta; \Gamma \vdash \mathit{mod} \Rightarrow \mathrm{M} : \mathrm{S}/(\Sigma; \delta)}$

$$
\frac{\begin{array}{c} \Theta; \Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow v : \tau/(\Sigma; \delta_1) \quad \mathsf{genvars}(\delta_1\Gamma; \Sigma; \tau) = \overline{\boldsymbol{\alpha}} \quad \mathsf{partition}(\overline{\boldsymbol{\alpha}}; \Sigma) = (\Sigma_1; \Sigma_2) \quad \mathsf{makesig}(\overline{\boldsymbol{\alpha}}; \Sigma_2) = \mathrm{S} \\ \overline{\boldsymbol{\alpha}} = \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m \quad \Sigma_2 = \mathrm{X}_1{:}\mathbb{S}_1, \ldots, \mathrm{X}_n{:}\mathbb{S}_n \quad \delta = \{\boldsymbol{\alpha}_i \mapsto \mathrm{X}.\mathsf{tyvars}.i\}_{i=1}^{m} \quad \sigma = \{\mathrm{X}_i \mapsto \mathrm{X}.\mathsf{consts}.i\}_{i=1}^{n} \end{array}}{\Theta; \Gamma \vdash [\mathit{exp}] \Rightarrow \Lambda \mathrm{X}{:}\mathrm{S}.[\sigma\delta v] : \forall \mathrm{X}{:}\mathrm{S}.\llbracket \delta\tau \rrbracket/(\Sigma_1; \delta_1)} \;(39)
$$

$$
\frac{\Theta; \Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma; \delta) \quad e \text{ not valuable}}{\Theta; \Gamma \vdash [\mathit{exp}] \Rightarrow [e] : \llbracket \tau \rrbracket/(\Sigma; \delta)} \;(40) \qquad \frac{\begin{array}{c} \Theta; \Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S} \quad \Gamma \vdash \mathrm{S}\ \mathsf{concrete} \\ \mathrm{S} \rightrightarrows \exists \overline{\boldsymbol{\alpha}}.\mathbb{S} \quad \Theta; \Gamma \vdash \mathrm{X}{:}\mathbb{S} \downarrow (\emptyset; \sigma; \delta) \quad \overline{\boldsymbol{\alpha}} \subseteq \mathrm{dom}(\delta) \end{array}}{\Theta; \Gamma \vdash \mathsf{canon}(\mathit{sig}) \Rightarrow \sigma\mathrm{X} : \delta\mathbb{S}/(\emptyset; \mathbf{id})} \;(41)
$$

$\boxed{\textbf{Constraint Normalization: } \Theta; \Gamma \vdash \Sigma_1 \downarrow (\Sigma_2; \sigma; \delta)}$

$$
\frac{\begin{array}{c} \forall \mathrm{X}{:}\mathbb{S} \in \Sigma.\ \exists \boldsymbol{\alpha}.\ \Gamma, \Sigma \vdash \mathrm{X}.\mathtt{t} \equiv \boldsymbol{\alpha} : \mathbf{T} \\ \forall \mathrm{X}_1{:}\mathbb{S}_1, \mathrm{X}_2{:}\mathbb{S}_2 \in \Sigma.\ (\mathrm{X}_1 \neq \mathrm{X}_2 \wedge \mathbb{S}_1 \approx \mathbb{S}_2) \Rightarrow \Gamma, \Sigma \vdash \mathrm{X}_1.\mathtt{t} \not\equiv \mathrm{X}_2.\mathtt{t} : \mathbf{T} \end{array}}{\Theta; \Gamma \vdash \Sigma \downarrow (\Sigma; \mathbf{id}; \mathbf{id})} \;(42) \qquad \frac{\begin{array}{c} \Theta; \Gamma \vdash \Sigma_1 \rightsquigarrow (\Sigma_2; \sigma_1; \delta_1) \\ \Theta; \delta_1\Gamma \vdash \Sigma_2 \downarrow (\Sigma_3; \sigma_2; \delta_2) \end{array}}{\Theta; \Gamma \vdash \Sigma_1 \downarrow (\Sigma_3; \sigma_2\delta_2\sigma_1; \delta_2\delta_1)} \;(43)
$$

$\boxed{\textbf{Constraint Reduction: } \Theta; \Gamma \vdash \Sigma_1 \rightsquigarrow (\Sigma_2; \sigma; \delta)}$

$$
\frac{}{\Theta; \Gamma \vdash \Sigma, \mathrm{X}{:}\llbracket \boldsymbol{\mathfrak{S}}(\tau) \rrbracket \rightsquigarrow (\Sigma; \{\mathrm{X} \mapsto [\tau]\}; \mathbf{id})} \;(44) \qquad \frac{\Gamma \vdash \mathbb{S} \downarrow \{\ell_1{:}\mathbb{S}_1, \ldots, \ell_n{:}\mathbb{S}_n\} \quad \mathtt{t} \notin \ell_1, \ldots, \ell_n}{\Theta; \Gamma \vdash \Sigma, \mathrm{X}{:}\mathbb{S} \rightsquigarrow (\Sigma, \mathrm{X}_1{:}\mathbb{S}_1, \ldots, \mathrm{X}_n{:}\mathbb{S}_n; \{\mathrm{X} \mapsto \{\ell_1{=}\mathrm{X}_1, \ldots, \ell_n{=}\mathrm{X}_n\}\}; \mathbf{id})} \;(45)
$$

$$
\frac{\begin{array}{c} \Gamma \vdash \mathbb{S} \downarrow \{\mathtt{t}{:}\llbracket \boldsymbol{\mathfrak{S}}(\tau) \rrbracket, \ldots\} \quad \tau \text{ not an } \boldsymbol{\alpha} \\ \mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} :\downarrow \mathbb{S}' \quad \Gamma \vdash \mathbb{S} \equiv \mathbb{S}' \Rightarrow \delta \end{array}}{\Theta; \Gamma \vdash \Sigma, \mathrm{X}{:}\mathbb{S} \rightsquigarrow (\delta\Sigma; \{\mathrm{X} \mapsto \mathrm{P}\}; \delta)} \;(46) \qquad \frac{\begin{array}{c} \Gamma \vdash \mathbb{S} \downarrow \{\mathtt{t}{:}\llbracket \boldsymbol{\mathfrak{S}}(\tau) \rrbracket, \ldots\} \quad \tau \text{ not an } \boldsymbol{\alpha} \\ \mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} :\downarrow \forall \mathrm{Y}{:}\mathrm{S}_1.\mathbb{S}_2 \quad \mathrm{S}_1 \rightrightarrows \exists \overline{\boldsymbol{\alpha}}.\mathbb{S}_1 \quad \Gamma, \mathrm{Y}{:}\mathbb{S}_1 \vdash \mathbb{S} \equiv \mathbb{S}_2 \Rightarrow \delta \end{array}}{\Theta; \Gamma \vdash \Sigma, \mathrm{X}{:}\mathbb{S} \rightsquigarrow (\delta\Sigma, \mathrm{Y}{:}\delta\mathbb{S}_1; \{\mathrm{X} \mapsto \mathrm{P}\langle \mathrm{Y} \rangle\}; \delta)} \;(47)
$$

$$
\frac{\exists \boldsymbol{\alpha}.\ \Gamma, \Sigma \vdash \mathrm{X}_1.\mathtt{t} \equiv \boldsymbol{\alpha} \equiv \mathrm{X}_2.\mathtt{t} : \mathbf{T} \quad \Gamma \vdash \mathbb{S}_1 \equiv \mathbb{S}_2 \Rightarrow \delta}{\Theta; \Gamma \vdash \Sigma, \mathrm{X}_1{:}\mathbb{S}_1, \mathrm{X}_2{:}\mathbb{S}_2 \rightsquigarrow (\delta\Sigma, \mathrm{X}_1{:}\delta\mathbb{S}_1; \{\mathrm{X}_2 \mapsto \mathrm{X}_1\}; \delta)} \;(48)
$$

Figure 4. Key Type Inference Rules

a canonical module of signature $\mathbb{S}$ to be substituted for X in the term or module that is output by elaboration.

Figure 4 consists of a selection of the most interesting rules in the type inference algorithm. We use $\Rightarrow$ instead of $\rightsquigarrow$ to distinguish the inference judgments from the elaboration judgments. The type unification judgment (Rule 31) first normalizes the given types, then performs syntactic unification on them. The latter is mostly standard, although in the base case of Rule 32 it is important to perform not only an occurs-check but a well-formedness check on the context. In the presence of explicit local abstract types, as arise for example from functor abstractions, an attempt may be made to unify $\boldsymbol{\alpha}$ with a type τ that is only valid in a later scope. The premise $\vdash \Gamma[\tau/\boldsymbol{\alpha}]$ ok safeguards against this.

Rules 33–35 are standard, but there are a few points of note. First, the $\boldsymbol{\alpha}$ that appears in these rules is implicitly chosen to be a fresh unification variable. Second, Rule 35 is representative of how most of the rules defining elaboration are easily converted into algorithmic rules by amassing Σ's and threading δ's. In the conclusion of the rule, the output substitution δ is subjected to an operation written $\delta|_\Gamma$. This restricts the domain of δ to $\mathrm{UV}(\Gamma)$ (the free unification variables of Γ). It is essentially a form of garbage collection that removes from δ any bindings for fresh unification variables that were introduced during the inference for $exp_1(exp_2)$. This step is useful for soundness purposes to ensure that the range of $\delta|_\Gamma$ is well-formed in the context $\delta\Gamma$.

The premises of these rules employ a variant of inference written with $\Rightarrow\downarrow$. This signifies the composition of inference with type and constraint normalization (see Rule 38). Constraint normalization takes an arbitrary constraint and reduces it to one in normal form. In a normal form constraint, every signature is atomic and its `t` component is equal to a unification variable. Certain rules (such as Rule 39 for generalization) require their premises to have their constraints normalized; most rules don't. Nevertheless, there is no harm in eagerly reducing all constraints to normal form. Constraint normalization is discussed in more detail below.

Rule 37 performs polymorphic instantiation. Given a path P of polymorphic signature $\forall \mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket$, it uses the judgment $\mathrm{S} \rightrightarrows \exists\overline{\boldsymbol{\alpha}}.\mathbb{S}$ to generate fresh unification variables $\overline{\boldsymbol{\alpha}}$ corresponding to the abstract type components of S. It then applies P to an unknown canonical module X of signature $\mathbb{S}$, and projects out the value component. This in turn effects a demand for $\mathrm{X}{:}\mathbb{S}$ in the output constraint. For example, if S were the class `EQ`, then the output constraint would be `X:EQ where type t =` $\boldsymbol{\alpha}$. (Note: the "$:\downarrow$" judgment used in the first premise indicates $\forall \mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket$ is the normal form signature of P, and the last premise normalizes τ so that references to type components of X become references to the corresponding $\overline{\boldsymbol{\alpha}}$.)

Rule 39 performs polymorphic generalization. The first premise translates exp to a valuable term v with type τ, and generates a normalized constraint Σ. The second premise calculates the largest set of variables $\overline{\boldsymbol{\alpha}}$ over which v may be abstracted. Based on $\overline{\boldsymbol{\alpha}}$, the third premise partitions Σ accordingly into Σ_2 (which will join $\overline{\boldsymbol{\alpha}}$ in the abstraction) and Σ_1 (which will propagate out of the rule). Essentially, Σ_2 comprises the constraints that refer to variables in $\overline{\boldsymbol{\alpha}}$ and Σ_1 comprises the constraints that do not. Finally, we use the makesig macro to combine $\overline{\boldsymbol{\alpha}}$ and Σ_2 into a class signature S that we can abstract v over. The remainder of the premises are simply doing namespace management to convert references to $\overline{\boldsymbol{\alpha}}$ and $\mathrm{dom}(\Sigma_2)$ into projections from the module variable X.

Rule 41 computes the canonical module of signature S by doing something similar to polymorphic instantiation. As in Rule 37, a constraint $\mathrm{X}{:}\mathbb{S}$ is constructed that fills in the abstract type components of S with fresh unification variables. However, since S is required to be concrete, these unification variables may only fill in associated type components. Thus, it must be the case that $\mathrm{X}{:}\mathbb{S}$ can be fully reduced via constraint normalization to the empty constraint. In the process, a canonical module substitution σ is generated such that $\sigma\mathrm{X}$ is canonical at signature $\delta\mathbb{S}$, a subtype of S.

As Rule 41 illustrates, the *constraint normalization* judgment is the place in the algorithm where canonical modules are actually computed. Normalization takes zero or more steps of *constraint reduction* until the input constraint is reduced to a normal form in which all residual constraints are instances of atomic classes at unification variables (Rules 42 and 43). The relation between the input and output of normalization is summarized by the following invariant:

$$\begin{gathered}\text{If } \Theta;\Gamma \vdash \Sigma_1 \downarrow (\Sigma_2;\sigma;\delta),\\ \text{then } \forall \mathrm{X}{:}\mathbb{S} \in \Sigma_1.\ \Theta, \mathrm{dom}(\Sigma_2);\delta\Gamma, \Sigma_2 \vdash_{\overline{\mathrm{can}}} \sigma\mathrm{X} : \delta\mathbb{S}.\end{gathered}$$

That is, if we treat the domain of the normalized constraint Σ_2 as a set of canonical instances, then from those instances together with the canonical instances already in Θ, the substitution σ shows how to construct canonical modules to satisfy all the demands of the original constraint Σ_1 (subject to type substitution δ).

To make this concrete, suppose Θ contains the `EqInt` and `EqProd` instance modules given in Section 2.1, and suppose that Σ_1 is X`:EQ where type t = int *` $\boldsymbol{\alpha}$. Then the normalized Σ_2 would be Y`:EQ where type t =` $\boldsymbol{\alpha}$, and the substitution σ would map X to `EqProd(EqInt,`Y`)`. (In this case, δ would simply be the identity substitution $\mathbf{id}$.)

Rules 44–47 for constraint reduction correspond closely to Haskell-style *context reduction*, aka *simplification*. In the terms of our elaborator, constraint reduction can be viewed as a backchaining implementation of the canonical module judgment. Rule 48 provides a form of constraint *improvement* [10]. If two constraints share their `t` component and their signatures are unifiable, then our definition of overlapping instances implies that the only way the constraints can possibly be satisfied is if they are unified into one.

Finally, we have not shown any inference rules for top-level modules because they are essentially identical to the corresponding elaboration rules. Since the outer level of the program is explicitly typed, there is no need at that level for any Damas-Milner-style type inference.

4.5 Soundness

We have proven that our inference algorithm is sound with respect to the elaboration semantics. We collect the main results here and refer to the appendix for the full statement (in Figure 23) and its auxiliary definitions (Figure 22), including the precise meaning of the theorem's preconditions.

Theorem (Soundness)
Suppose $(\Theta;\Gamma)$ is valid for inference, $\Theta' \supseteq \Theta$, $\Gamma' \vdash \delta' : \delta\Gamma$, $\vdash (\Theta';\Gamma')$ ok, and $\forall \mathrm{X}{:}\mathbb{S} \in \Sigma.\ \Theta';\Gamma' \vdash_{\overline{\mathrm{can}}} \sigma'\mathrm{X} : \delta'\mathbb{S}$. Then:

1. If $\Theta;\Gamma \vdash exp \Rightarrow e : \tau/(\Sigma;\delta)$,
   then $\Theta';\Gamma' \vdash exp \rightsquigarrow \sigma'\delta' e : \delta'\tau$.
2. If $\Theta;\Gamma \vdash mod \Rightarrow \mathrm{M} : \mathrm{S}/(\Sigma;\delta)$,
   then $\Theta';\Gamma' \vdash mod \rightsquigarrow \sigma'\delta'\mathrm{M} : \delta'\mathrm{S}$.

Consider part 1. Informally, Θ, Γ and exp are inputs. If type inference on exp succeeds, it produces an IL term e, along with a constraint Σ and a substitution δ. If in any "future world" $(\Theta';\Gamma')$ the constraint Σ can be solved by substitutions σ' and δ', then exp will declaratively elaborate to $\sigma'\delta' e$ in that world. The theorem statement for modules (part 2) is analogous.

4.6 Incompleteness

While the inference algorithm is sound, it is *not* complete, for reasons that arise independently of the present work. One source of incompleteness is inherited from Haskell and concerns a fundamental problem with type classes, namely the problem of *ambiguity* [14]. The canonical example uses the following two signatures:

```
signature SHOW = sig
  type t
  val show : t -> string
end
signature READ = sig
  type t
  val read : string -> t
end
val show = overload show from SHOW
val read = overload read from READ
```

Given this overloading, the expression `show (read ("1"))` is ambiguous, as the result type of `read` and argument type of `show` are completely unconstrained. This is problematic because, depending on the available canonical instances, two or more valid elaborations with observably different behaviour may exist. Hence, ambiguous programs need to be rejected. This can be done easily during inference, but for inference to be complete the completeness theorem has to be formulated in such a way that ambiguous programs are excluded from consideration. We have avoided this issue here entirely in the interest of a clearer presentation.

Another source of incompleteness is inherited from ML, and arises from the interaction between modules and type inference. Consider the following Standard ML program:

```
functor F(X : sig type t end) = struct
  val f = (print "Hello"; fn x => x)
end
structure Y1 = F(struct type t = int end)
structure Y2 = F(struct type t = bool end)
val z1 = Y1.f(3)
val z2 = Y2.f(true)
```

The binding of `f` in `F` is chosen to have an effect, so that it cannot be given a polymorphic type. This raises the question of what signature should be assigned to `F`. According to the Definition of Standard ML [18] (and the HS semantics as well), the above program is well-typed because `f` may be assigned the type `X.t -> X.t`, which is consistent with both subsequent uses of `F`. But in order to figure this out, a compiler would have to do a form of higher-order unification—once we leave the scope of `X.t`, the unification variable in the type of `f` should be skolemized over `X.t`.

As a result, nearly all existing implementations of Standard ML reject this program, as do we. (The only one that accepts it is MLton, but MLton also *accepts* similar programs that the Definition *rejects* [5].) This example points out that the interactions between type inference and modules are still not fully understood, and merit further investigation beyond the scope of this paper.

5. Related Work

Type classes in Haskell. Since Wadler and Blott's seminal paper [25], the basic system of type classes has been extended in a number of ways. Of these, Jones' framework of *qualified types* [11] and the resulting generalizations to constructor classes [15], multi-parameter type classes, and functional dependencies [12] are the most widely used. We discussed the option of supporting multi-parameter and constructor classes in the modular setting in Section 3.4. Instead of functional dependencies, we support associated types, as they arise naturally from type components in modules.

Achieving a separation between instance declaration and instance adoption, so that instance declarations need not have global scope, is still an open problem in the Haskell setting. There exists an experimental proposal by Kahl and Scheffczyk [16] that is motivated by a comparison with ML modules. Their basic idea is to allow constrained polymorphic functions to be given explicit instance arguments instead of having their instance arguments computed automatically. We support this functionality by providing the ability to coerce back and forth between polymorphic functions and functors, the latter of which may be given explicit module arguments (Section 2.4). Moreover, we permit different instances of the same signature to be made canonical in different scopes, which Kahl and Scheffczyk do not.

Comparing type classes and modules. The only formal comparison between ML modules and Haskell type classes is by Wehr [26]. He formalizes a translation from type classes to modules and vice versa, proves that both translations are type-preserving, and uses the translations as the basis for a comparison of the expressiveness of the language features. Wehr concludes that his encoding can help Haskell programmers to emulate certain aspects of modules in Haskell, but that the module encoding of type classes in ML is too heavyweight to be used for realistic programs. Not surprisingly, Wehr's encoding of type classes as modules uses signatures for classes and modules for instances, as we do. In fact, his translation can be regarded as an elaboration from a Haskell core language to a fragment of ML. However, the fundamental difference between our work and his is that he performs elaboration in the *non-modular* context of Haskell, whereas we demonstrate how to perform elaboration and type inference in the *modular* context of ML.

Type classes for ML. Schneider [20] has proposed to extend ML with type classes as a feature independent of modules. This leads to significant duplication of mechanism and a number of technical problems, which we avoid by expressing type classes via modules. More recently, Siek and Lumsdaine [22] have described a language F^{G} that integrates *concepts*, which are closely related to type classes, into System F. However, F^{G} does not support type inference. Siek's thesis [21] defines a related language $\mathcal{G}$, which supports inference for type applications, but not type abstractions. Concepts in $\mathcal{G}$ are treated as a distinct construct, unrelated to modules, and $\mathcal{G}$ does not support parameterized modules (*i.e.*, functors).

Parameterized signatures. Jones [13] has proposed a way of supporting modular programming in a Haskell-like language, in which a signature is encoded as a record type parameterized over the abstract type components of the signature. However, he does not consider the interaction with type classes.

Acknowledgments

We thank Stefan Wehr for stimulating discussions on ways of representing type classes with modules.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05*.

[2] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05*.

[3] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *POPL '82*.

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

[5] Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. Technical Report TR-2006-08, University of Chicago Comp. Sci. Dept., October 2006.

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL '03*.

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

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

[9] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof,*

Language, and Interaction: Essays in Honor of Robin Milner. MIT Press, 2000.

[10] Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95*.

[11] Mark P. Jones. A theory of qualified types. In *ESOP '92*.

[12] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00*.

[13] Mark P. Jones. Using parameterized signatures to express modular structure. In *POPL '96*.

[14] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[15] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), 1995.

[16] Wolfram Kahl and Jan Scheffczyk. Named instances for Haskell type classes. In *Haskell Workshop*, 2001.

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

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

[19] Simon Peyton Jones et al. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), 2003.

[20] Gerhard Schneider. ML mit Typklassen. Master's thesis, June 2000.

[21] Jeremy Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005.

[22] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05*.

[23] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *POPL '00*.

[24] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 2006. To appear.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL '89*.

[26] Stefan Wehr. ML modules and Haskell type classes: A constructive comparison. Master's thesis, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2005.

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathsf{Fst}(\llbracket \mathrm{K} \rrbracket) & \stackrel{\mathrm{def}}{=} & \mathrm{K} & \boldsymbol{\mathfrak{S}}_{\llbracket \mathrm{K} \rrbracket}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \llbracket \boldsymbol{\mathfrak{S}}_{\mathrm{K}}(\mathrm{C}) \rrbracket \\
\mathsf{Fst}(\llbracket \tau \rrbracket) & \stackrel{\mathrm{def}}{=} & \{\} & \boldsymbol{\mathfrak{S}}_{\llbracket \tau \rrbracket}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \llbracket \tau \rrbracket \\
\mathsf{Fst}(\{\overline{\ell \rhd \mathrm{X}{:}\mathrm{S}}\}) & \stackrel{\mathrm{def}}{=} & \{\overline{\ell \rhd \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S})}\} & \boldsymbol{\mathfrak{S}}_{\{\overline{\ell \rhd \mathrm{X}:\mathrm{S}}\}}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \{\overline{\ell \rhd \mathrm{X}{:}\boldsymbol{\mathfrak{S}}_{\mathrm{S}}(\mathrm{C}.\ell)}\} \\
\mathsf{Fst}(\forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2) & \stackrel{\mathrm{def}}{=} & \Pi \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1).\mathsf{Fst}(\mathbb{S}_2) & \boldsymbol{\mathfrak{S}}_{\forall \mathrm{X}:\mathrm{S}_1.\mathbb{S}_2}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \\
\mathsf{Fst}(\Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2) & \stackrel{\mathrm{def}}{=} & \{\} & \boldsymbol{\mathfrak{S}}_{\Pi \mathrm{X}:\mathrm{S}_1.\mathrm{S}_2}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2
\end{array}$$

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathsf{Fst}(\mathrm{X}) & \stackrel{\mathrm{def}}{=} & \mathrm{X}^{\mathsf{c}} & \mathsf{Fst}(\mathbb{M}.\ell) & \stackrel{\mathrm{def}}{=} & \mathsf{Fst}(\mathbb{M}).\ell \\
\mathsf{Fst}([\mathrm{C}]) & \stackrel{\mathrm{def}}{=} & \mathrm{C} & \mathsf{Fst}(\Lambda \mathrm{X}{:}\mathrm{S}.\mathbb{M}) & \stackrel{\mathrm{def}}{=} & \lambda \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}).\mathsf{Fst}(\mathbb{M}) \\
\mathsf{Fst}([e]) & \stackrel{\mathrm{def}}{=} & \{\} & \mathsf{Fst}(\mathbb{F}\langle \mathbb{M} \rangle) & \stackrel{\mathrm{def}}{=} & \mathsf{Fst}(\mathbb{F})(\mathsf{Fst}(\mathbb{M})) \\
\mathsf{Fst}(\{\overline{\ell \rhd \mathrm{X}{=}\mathbb{M}}\}) & \stackrel{\mathrm{def}}{=} & \{\overline{\ell \rhd \mathrm{X}^{\mathsf{c}}{=}\mathsf{Fst}(\mathbb{M})}\} & \mathsf{Fst}(\lambda(\mathrm{X}{:}\mathrm{S}_1)\,{:}{>}\mathrm{S}_2.\mathrm{M}) & \stackrel{\mathrm{def}}{=} & \{\}
\end{array}$$

Notation: $\boldsymbol{\mathfrak{S}}_{\mathrm{S}}(\mathbb{M})$ and $E[\mathbb{M}/\mathrm{X}]$ are shorthand for $\boldsymbol{\mathfrak{S}}_{\mathrm{S}}(\mathsf{Fst}(\mathbb{M}))$ and $E[\mathsf{Fst}(\mathbb{M})/\mathrm{X}^{\mathsf{c}}]$, respectively.

Well-formed Contexts: $\vdash \Gamma\ \mathsf{ok}$

$$\frac{}{\vdash \emptyset\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \mathrm{K}\ \mathrm{kind}}{\vdash \Gamma, \alpha{:}\mathrm{K}\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \tau : \mathbf{T}}{\vdash \Gamma, x{:}\tau\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \mathrm{S}\ \mathrm{sig}}{\vdash \Gamma, \mathrm{X}{:}\mathrm{S}\ \mathsf{ok}}$$

Kind Subtyping: $\Gamma \vdash \mathrm{K}_1 \leq \mathrm{K}_2$

$$\frac{}{\Gamma \vdash \mathbf{T} \leq \mathbf{T}} \qquad \frac{\Gamma \vdash \tau : \mathbf{T}}{\Gamma \vdash \boldsymbol{\mathfrak{S}}(\tau) \leq \mathbf{T}} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 : \mathbf{T}}{\Gamma \vdash \boldsymbol{\mathfrak{S}}(\tau_1) \leq \boldsymbol{\mathfrak{S}}(\tau_2)} \qquad \frac{\Gamma \vdash \mathrm{K}_1' \equiv \mathrm{K}_2' \quad \Gamma, \alpha{:}\mathrm{K}_1' \vdash \mathrm{K}_1'' \leq \mathrm{K}_2''}{\Gamma \vdash \Pi \alpha{:}\mathrm{K}_1'.\mathrm{K}_1'' \leq \Pi \alpha{:}\mathrm{K}_2'.\mathrm{K}_2''}$$

$$\frac{}{\Gamma \vdash \{\} \leq \{\}} \qquad \frac{\Gamma \vdash \mathrm{K}_1 \leq \mathrm{K}_1' \quad \Gamma, \alpha{:}\mathrm{K}_1 \vdash \{\overline{\ell \rhd \alpha{:}\mathrm{K}}\} \leq \{\overline{\ell \rhd \alpha{:}\mathrm{K}'}\} \quad \Gamma \vdash \{\ell_1 \rhd \alpha_1{:}\mathrm{K}_1', \overline{\ell \rhd \alpha{:}\mathrm{K}'}\}\ \mathrm{kind}}{\Gamma \vdash \{\ell_1 \rhd \alpha_1{:}\mathrm{K}_1, \overline{\ell \rhd \alpha{:}\mathrm{K}}\} \leq \{\ell_1 \rhd \alpha_1{:}\mathrm{K}_1', \overline{\ell \rhd \alpha{:}\mathrm{K}'}\}}$$

Signature Subtyping: $\Gamma \vdash \mathrm{S}_1 \leq \mathrm{S}_2$

$$\frac{\Gamma \vdash \mathrm{K}_1 \leq \mathrm{K}_2}{\Gamma \vdash \llbracket \mathrm{K}_1 \rrbracket \leq \llbracket \mathrm{K}_2 \rrbracket} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 : \mathbf{T}}{\Gamma \vdash \llbracket \tau_1 \rrbracket \leq \llbracket \tau_2 \rrbracket} \qquad \frac{\Gamma \vdash \mathrm{S}_1' \equiv \mathrm{S}_2' \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1' \vdash \mathrm{S}_1'' \equiv \mathrm{S}_2''}{\Gamma \vdash \Pi \mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_1'' \leq \Pi \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''} \qquad \frac{\Gamma \vdash \mathrm{S}_1' \equiv \mathrm{S}_2' \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1' \vdash \mathbb{S}_1'' \equiv \mathbb{S}_2''}{\Gamma \vdash \forall \mathrm{X}{:}\mathrm{S}_1'.\mathbb{S}_1'' \leq \forall \mathrm{X}{:}\mathrm{S}_2'.\mathbb{S}_2''}$$

$$\frac{}{\Gamma \vdash \{\} \leq \{\}} \qquad \frac{\Gamma \vdash \mathrm{S}_1 \leq \mathrm{S}_1' \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \{\overline{\ell \rhd \mathrm{X}{:}\mathrm{S}}\} \leq \{\overline{\ell \rhd \mathrm{X}{:}\mathrm{S}'}\} \quad \Gamma \vdash \{\ell_1 \rhd \mathrm{X}_1{:}\mathrm{S}_1', \overline{\ell \rhd \mathrm{X}{:}\mathrm{S}'}\}\ \mathrm{sig}}{\Gamma \vdash \{\ell_1 \rhd \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \rhd \mathrm{X}{:}\mathrm{S}}\} \leq \{\ell_1 \rhd \mathrm{X}_1{:}\mathrm{S}_1', \overline{\ell \rhd \mathrm{X}{:}\mathrm{S}'}\}}$$

The judgments $\Gamma \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$ and $\Gamma \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$ are defined to coincide with subtyping in both directions.
For details of the kinding and equivalence judgments for type constructors, see Dreyer's thesis [4].

Figure 5. IL Static Semantics (Abridged)

$\boxed{\textbf{Kinds:}\ \Theta;\Gamma \vdash \mathit{knd} \rightsquigarrow \mathrm{K}}$

$$\frac{\mathit{knd} = \mathbf{T} \text{ or } \mathbf{T}^n \rightarrow \mathbf{T}}{\Theta;\Gamma \vdash \mathit{knd} \rightsquigarrow \mathit{knd}} \qquad \frac{\Theta;\Gamma \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T}}{\Theta;\Gamma \vdash \mathfrak{S}(\mathit{typ}) \rightsquigarrow \mathfrak{S}(\tau)} \qquad \frac{\Theta;\Gamma,\overline{\alpha} \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T} \quad \overline{\alpha} = \alpha_1,\ldots,\alpha_n}{\Theta;\Gamma \vdash \Pi\overline{\alpha}.\mathfrak{S}(\mathit{typ}) \rightsquigarrow \Pi\alpha{:}\mathbf{T}^n.\mathfrak{S}(\mathit{typ}[\alpha.i/\alpha_i])}$$

$\boxed{\textbf{Type Constructors:}\ \Theta;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{C} : \mathrm{K}}$

$$\frac{\alpha{:}\mathrm{K} \in \Gamma}{\Theta;\Gamma \vdash \alpha \rightsquigarrow \alpha : \mathrm{K}} \qquad \frac{\forall i \in \{1,2\}.\ \Theta;\Gamma \vdash \mathit{typ}_i \rightsquigarrow \tau_i : \mathbf{T}}{\Theta;\Gamma \vdash \mathit{typ}_1 \rightarrow \mathit{typ}_2 \rightsquigarrow \tau_1 \rightarrow \tau_2 : \mathbf{T}} \qquad \frac{\Gamma \vdash \mathrm{P} : [\![\mathrm{K}]\!]}{\Theta;\Gamma \vdash \mathrm{P} \rightsquigarrow \mathsf{Fst}(\mathrm{P}) : \mathrm{K}}$$

$$\frac{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \rightsquigarrow \mathbb{V} : \mathbb{S} \quad \Gamma \vdash \mathbb{V}.\ell s : [\![\mathrm{K}]\!]}{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}).\ell s \rightsquigarrow \mathsf{Fst}(\mathbb{V}).\ell s : \mathrm{K}} \qquad \frac{\Theta;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{C} : \mathrm{K}' \quad \Gamma \vdash \mathrm{C} : \mathrm{K}}{\Theta;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{C} : \mathrm{K}}$$

$$\frac{\Theta;\Gamma,\overline{\alpha} \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T} \quad \overline{\alpha} = \alpha_1,\ldots,\alpha_n}{\Theta;\Gamma \vdash \lambda\overline{\alpha}.\mathit{typ} \rightsquigarrow \lambda\alpha{:}\mathbf{T}^n.\tau[\alpha.i/\alpha_i] : \mathbf{T}^n \rightarrow \mathbf{T}} \qquad \frac{\Theta;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{C} : \mathbf{T}^n \rightarrow \mathbf{T} \quad \forall i \in 1..n.\ \Theta;\Gamma \vdash \mathit{typ}_i \rightsquigarrow \tau_i : \mathbf{T}}{\Theta;\Gamma \vdash \mathit{con}(\mathit{typ}_1,\ldots,\mathit{typ}_n) \rightsquigarrow \mathrm{C}(\tau_1,\ldots,\tau_n) : \mathbf{T}}$$

$\boxed{\textbf{Signatures:}\ \Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S}}$

$$\frac{\Theta;\Gamma \vdash \mathit{knd} \rightsquigarrow \mathrm{K}}{\Theta;\Gamma \vdash [\![\mathit{knd}]\!] \rightsquigarrow [\![\mathrm{K}]\!]} \qquad \frac{\Theta;\Gamma \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T}}{\Theta;\Gamma \vdash [\![\mathit{typ}]\!] \rightsquigarrow [\![\tau]\!]} \qquad \frac{}{\Theta;\Gamma \vdash \{\} \rightsquigarrow \{\}} \qquad \frac{\Theta;\Gamma \vdash \mathit{sig}_1 \rightsquigarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \{\overline{\ell \triangleright \mathrm{X}{:}\mathit{sig}}\} \rightsquigarrow \{\overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\}}{\Theta;\Gamma \vdash \{\ell_1 \triangleright \mathrm{X}_1{:}\mathit{sig}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathit{sig}}\} \rightsquigarrow \{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\}}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig}_1 \rightsquigarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{sig}_2 \rightsquigarrow \mathrm{S}_2 \quad \mathrm{S}_2 = \{\ldots\}}{\Theta;\Gamma \vdash \Pi\mathrm{X}{:}\mathit{sig}_1.\mathit{sig}_2 \rightsquigarrow \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2} \qquad \frac{\Theta;\Gamma \vdash \mathit{sig}_1 \rightsquigarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{tsig}_2 \rightsquigarrow \mathbb{S}_2 \quad \mathbb{S}_2 = \{\ldots\}}{\Theta;\Gamma \vdash \forall\mathrm{X}{:}\mathit{sig}_1.\mathit{tsig}_2 \rightsquigarrow \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S} \quad \Gamma \vdash_{\overline{\text{class}}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta' \quad \Theta,\Theta';\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T}}{\Theta;\Gamma \vdash \forall\mathrm{X}{:}\mathit{sig}.[\![\mathit{typ}]\!] \rightsquigarrow \forall\mathrm{X}{:}\mathrm{S}.[\![\tau]\!]}$$

Figure 6. Elaboration Rules for Kinds, Types, and Signatures

$\boxed{\textbf{Terms:}\ \Theta;\Gamma \vdash \mathit{exp} \rightsquigarrow e : \tau}$

$$\frac{x{:}\tau \in \Gamma}{\Theta;\Gamma \vdash x \rightsquigarrow x : \tau} \qquad \frac{\Gamma \vdash \tau_1 : \mathbf{T} \quad \Theta;\Gamma,x{:}\tau_1 \vdash \mathit{exp} \rightsquigarrow e : \tau_2}{\Theta;\Gamma \vdash \lambda x.\mathit{exp} \rightsquigarrow \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Theta;\Gamma \vdash \mathit{exp}_1 \rightsquigarrow e_1 : \tau_2 \rightarrow \tau \quad \Theta;\Gamma \vdash \mathit{exp}_2 \rightsquigarrow e_2 : \tau_2}{\Theta;\Gamma \vdash \mathit{exp}_1(\mathit{exp}_2) \rightsquigarrow e_1(e_2) : \tau}$$

$$\frac{\Gamma \vdash \mathrm{P} : [\![\tau]\!]}{\Theta;\Gamma \vdash \mathrm{P} \rightsquigarrow \mathsf{Val}(\mathrm{P}) : \tau} \qquad \frac{\Gamma \vdash \mathrm{P} : \forall\mathrm{X}{:}\mathrm{S}.[\![\tau]\!] \quad \Gamma \vdash \mathbb{S} \leq \mathrm{S} \quad \Theta;\Gamma \vdash_{\overline{\text{can}}} \mathbb{V} : \mathbb{S}}{\Theta;\Gamma \vdash \mathrm{P} \rightsquigarrow \mathsf{Val}(\mathrm{P}\langle\mathbb{V}\rangle) : \tau[\mathbb{V}/\mathrm{X}]}$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod} \rightsquigarrow \mathrm{M} : \mathrm{S} \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathit{exp} \rightsquigarrow e : \tau \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\tau)}{\Theta;\Gamma \vdash \mathsf{let}\,\mathrm{X}{=}\mathit{mod}\,\mathsf{in}\ \mathit{exp} \rightsquigarrow \mathsf{let}\,\mathrm{X}{=}\mathrm{M}\,\mathsf{in}\ e : \tau}$$

$$\frac{\Theta;\Gamma \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T} \quad \Theta;\Gamma \vdash \mathit{exp} \rightsquigarrow e : \tau}{\Theta;\Gamma \vdash \mathit{exp} : \mathit{typ} \rightsquigarrow e : \tau} \qquad \frac{\Theta;\Gamma \vdash \mathit{exp} \rightsquigarrow e : \tau' \quad \Gamma \vdash \tau' \equiv \tau : \mathbf{T}}{\Theta;\Gamma \vdash \mathit{exp} \rightsquigarrow e : \tau}$$

Figure 7. Elaboration Rules for Terms

Modules: $\Theta;\Gamma \vdash \mathit{mod} \rightsquigarrow \mathrm{M} : \mathrm{S}$

$$\frac{\Gamma \vdash \mathrm{P} : \mathbb{S}}{\Theta;\Gamma \vdash \mathrm{P} \rightsquigarrow \mathrm{P} : \mathbb{S}} \qquad \frac{\Theta;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{C} : \mathbb{K}}{\Theta;\Gamma \vdash [\mathit{con}] \rightsquigarrow [\mathrm{C}] : \llbracket \mathbb{K} \rrbracket} \qquad \frac{\Theta;\Gamma \vdash \mathit{exp} \rightsquigarrow e : \tau}{\Theta;\Gamma \vdash [\mathit{exp}] \rightsquigarrow [e] : \llbracket \tau \rrbracket}$$

$$\frac{\mathrm{X} \notin \mathrm{FV}(\mathit{exp}) \quad \Gamma \vdash_{\mathrm{class}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta' \quad \Theta,\Theta';\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathit{exp} \rightsquigarrow v : \tau}{\Theta;\Gamma \vdash [\mathit{exp}] \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}.[v] : \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket}$$

$$\frac{}{\Theta;\Gamma \vdash \{\} \rightsquigarrow \{\} : \{\}} \qquad \frac{\Theta;\Gamma \vdash \mathit{mod}_1 \rightsquigarrow \mathrm{M}_1 : \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \{\overline{\ell \triangleright \mathrm{X}{=}\mathit{mod}}\} \rightsquigarrow \{\overline{\ell \triangleright \mathrm{X}{=}\mathrm{M}}\} : \{\overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\}}{\Theta;\Gamma \vdash \{\ell_1 \triangleright \mathrm{X}_1{=}\mathit{mod}_1, \overline{\ell \triangleright \mathrm{X}{=}\mathit{mod}}\} \rightsquigarrow \{\ell_1 \triangleright \mathrm{X}_1{=}\mathrm{M}_1, \overline{\ell \triangleright \mathrm{X}{=}\mathrm{M}}\} : \{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\}}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod} \rightsquigarrow \mathrm{M} : \mathrm{S}_2 \quad \mathrm{S}_2 = \{\ldots\}}{\Theta;\Gamma \vdash \lambda\mathrm{X}{:}\mathit{sig}.\mathit{mod} \rightsquigarrow \lambda(\mathrm{X}{:}\mathrm{S}_1) \mathbin{:>} \mathrm{S}_2.\mathrm{M} : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2} \qquad \frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod} \rightsquigarrow \mathrm{V} : \mathbb{S}_2 \quad \mathbb{S}_2 = \{\ldots\}}{\Theta;\Gamma \vdash \Lambda\mathrm{X}{:}\mathit{sig}.\mathit{mod} \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}_1.\mathrm{V} : \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2}$$

$$\frac{\Gamma \vdash \mathrm{P}_1 : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S} \quad \mathrm{S} = \{\ldots\} \quad \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \quad \Theta;\Gamma \vdash \mathrm{P}_2 \preceq \mathrm{S}_1 \rightsquigarrow \mathbb{V} : \mathbb{S}_1}{\Theta;\Gamma \vdash \mathrm{P}_1(\mathrm{P}_2) \rightsquigarrow \mathrm{P}_1(\mathbb{V}) : \mathrm{S}[\mathbb{V}/\mathrm{X}]}$$

$$\frac{\Gamma \vdash \mathrm{P}_1 : \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S} \quad \mathbb{S} = \{\ldots\} \quad \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \quad \Theta;\Gamma \vdash \mathrm{P}_2 \preceq \mathrm{S}_1 \rightsquigarrow \mathbb{V} : \mathbb{S}_1}{\Theta;\Gamma \vdash \mathrm{P}_1\langle\mathrm{P}_2\rangle \rightsquigarrow \mathrm{P}_1\langle\mathbb{V}\rangle : \mathbb{S}[\mathbb{V}/\mathrm{X}]}$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod}_1 \rightsquigarrow \mathrm{M}_1 : \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod}_2 \rightsquigarrow \mathrm{M}_2 : \mathrm{S} \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathrm{S})}{\Theta;\Gamma \vdash \mathsf{let}\,\mathrm{X}{=}\mathit{mod}_1\;\mathsf{in}\;\mathit{mod}_2 \rightsquigarrow \mathsf{let}\,\mathrm{X}{=}\mathrm{M}_1\;\mathsf{in}\;\mathrm{M}_2 \mathbin{:>} \mathrm{S} : \mathrm{S}}$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod} \rightsquigarrow \mathrm{M} : \mathrm{S}_1 \quad \Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S} \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{S} \rightsquigarrow \mathbb{V} : \mathbb{S}}{\Theta;\Gamma \vdash \mathit{mod} \mathbin{:>} \mathit{sig} \rightsquigarrow \mathsf{let}\,\mathrm{X}{=}\mathrm{M}\;\mathsf{in}\;\mathbb{V} \mathbin{:>} \mathrm{S} : \mathrm{S}} \qquad \frac{\Theta;\Gamma \vdash \mathit{mod} \rightsquigarrow \mathrm{M} : \mathrm{S}' \quad \Gamma \vdash \mathrm{S}' \equiv \mathrm{S}}{\Theta;\Gamma \vdash \mathit{mod} \rightsquigarrow \mathrm{M} : \mathrm{S}}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S} \quad \Gamma \vdash \mathrm{S}\;\mathsf{concrete} \quad \Gamma \vdash \mathbb{S} \leq \mathrm{S} \quad \Theta;\Gamma \vdash_{\mathrm{can}} \mathbb{V} : \mathbb{S}}{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \rightsquigarrow \mathbb{V} : \mathbb{S}} \qquad \frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S} \quad \Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathrm{X}.\ell s : \llbracket \tau \rrbracket \quad \Gamma \vdash \mathrm{S}\;\mathsf{class}}{\Theta;\Gamma \vdash \mathsf{overload}\;\ell s\;\mathsf{from}\;\mathit{sig} \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{X}.\ell s) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathrm{S}_1 \quad \Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \mathrm{X}_1.\ell s : \forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket \tau \rrbracket \quad \mathrm{S} = \{1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, 2 \triangleright \mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\;\mathsf{class}}{\Theta;\Gamma \vdash \mathsf{overload}\;\ell s\;\mathsf{from}\;\mathit{sig} \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{X}.1.\ell s\langle\mathrm{X}.2\rangle) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau[\mathrm{X}.i/\mathrm{X}_i] \rrbracket}$$

$$\frac{\Gamma \vdash \mathrm{P} : \forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket \tau \rrbracket\} \quad \Gamma \vdash \mathrm{S}\;\mathsf{class}}{\Theta;\Gamma \vdash \mathsf{implicit}(\mathrm{P}) \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{P}\langle\mathrm{X}\rangle.\mathsf{it}) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket}$$

$$\frac{\Gamma \vdash \mathrm{P} : \forall\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{:}\forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket \tau \rrbracket\} \quad \mathrm{S} = \{1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, 2 \triangleright \mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\;\mathsf{class}}{\Theta;\Gamma \vdash \mathsf{implicit}(\mathrm{P}) \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{P}\langle\mathrm{X}.1\rangle.\mathsf{it}\langle\mathrm{X}.2\rangle) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau[\mathrm{X}.i/\mathrm{X}_i] \rrbracket}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket \tau \rrbracket\} \quad \Gamma \vdash \mathrm{S}\;\mathsf{class} \quad \Theta;\Gamma \vdash \mathrm{P} \preceq \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket \rightsquigarrow \mathbb{V} : \_}{\Theta;\Gamma \vdash \mathsf{explicit}(\mathrm{P} : \mathit{sig}) \rightsquigarrow \Lambda\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{=}\mathbb{V}\langle\mathrm{X}\rangle\} : \forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket \tau \rrbracket\}}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \rightsquigarrow \forall\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{:}\forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket \tau \rrbracket\} \quad \mathrm{S} = \{1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, 2 \triangleright \mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\;\mathsf{class} \quad \Theta;\Gamma \vdash \mathrm{P} \preceq \forall\mathrm{X}{:}\mathrm{S}.\llbracket \tau[\mathrm{X}.i/\mathrm{X}_i] \rrbracket \rightsquigarrow \mathbb{V} : \_}{\Theta;\Gamma \vdash \mathsf{explicit}(\mathrm{P} : \mathit{sig}) \rightsquigarrow \Lambda\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{=}\Lambda\mathrm{X}_2{:}\mathrm{S}_2.\mathbb{V}\langle\{1{=}\mathrm{X}_1, 2{=}\mathrm{X}_2\}\rangle\} : \forall\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{:}\forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket \tau \rrbracket\}}$$

Figure 8. Elaboration Rules for Modules

Coercive Signature Matching: $\Theta; \Gamma \vdash \mathrm{P} \preceq \mathrm{S} \leadsto \mathbb{V} : \mathbb{S}$

$$\frac{\Gamma \vdash \mathrm{P} : \llbracket \mathbb{K} \rrbracket \quad \Gamma \vdash \mathbb{K} \leq \mathrm{K}}{\Theta; \Gamma \vdash \mathrm{P} \preceq \llbracket \mathrm{K} \rrbracket \leadsto \mathrm{P} : \llbracket \mathbb{K} \rrbracket} \qquad \frac{\mathbb{S} \text{ is of the form } \llbracket \tau \rrbracket \text{ or } \forall \mathrm{X}{:}\mathrm{S}.\llbracket \tau \rrbracket \quad \Theta; \Gamma \vdash [\mathrm{P}] \leadsto \mathbb{V} : \mathbb{S}}{\Theta; \Gamma \vdash \mathrm{P} \preceq \mathbb{S} \leadsto \mathbb{V} : \mathbb{S}}$$

$$\frac{\Gamma \vdash \mathrm{P} : \{\ldots\}}{\Theta; \Gamma \vdash \mathrm{P} \preceq \{\} \leadsto \{\} : \{\}} \qquad \frac{\Gamma \vdash \mathrm{P}.\ell_1 : \mathrm{S}_1' \quad \Theta; \Gamma \vdash \mathrm{P}.\ell_1 \preceq \mathrm{S}_1 \leadsto \mathbb{V}_1 : \mathbb{S}_1 \quad \Theta; \Gamma, \mathrm{X}_1{:}\mathbb{S}_1 \vdash \mathrm{P} \preceq \{\overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\} \leadsto \{\overline{\ell \triangleright \mathrm{X}{=}\mathbb{V}}\} : \{\overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}}{\Theta; \Gamma \vdash \mathrm{P} \preceq \{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\} \leadsto \{\ell_1 \triangleright \mathrm{X}_1{=}\mathbb{V}_1, \overline{\ell \triangleright \mathrm{X}{=}\mathbb{V}}\} : \{\ell_1 \triangleright \mathrm{X}_1{:}\mathbb{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}}$$

$$\frac{\mathbb{S}_2 = \{\ldots\} \quad \Gamma \vdash \mathrm{P} : \forall \mathrm{Y}{:}\mathrm{R}_1.\mathbb{R}_2 \quad \Theta; \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{R}_1 \leadsto \mathbb{V}_1 : \_ \quad \Theta; \Gamma, \mathrm{X}{:}\mathrm{S}_1, \mathrm{Z}{:}\mathbb{R}_2[\mathbb{V}_1/\mathrm{Y}] \vdash \mathrm{Z} \preceq \mathbb{S}_2 \leadsto \mathbb{V}_2 : \_}{\Theta; \Gamma \vdash \mathrm{P} \preceq \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \leadsto \Lambda \mathrm{X}{:}\mathrm{S}_1.\{1 \triangleright \mathrm{Z}{=}\mathrm{P}\langle \mathbb{V}_1 \rangle, 2{=}\mathbb{V}_2\}.2 : \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2}$$

$$\frac{\mathrm{S}_2 = \{\ldots\} \quad \Gamma \vdash \mathrm{P} : \forall \mathrm{Y}{:}\mathrm{R}_1.\mathbb{R}_2 \quad \Theta; \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{R}_1 \leadsto \mathbb{V}_1 : \_ \quad \Theta; \Gamma, \mathrm{X}{:}\mathrm{S}_1, \mathrm{Z}{:}\mathbb{R}_2[\mathbb{V}_1/\mathrm{Y}] \vdash \mathrm{Z} \preceq \mathrm{S}_2 \leadsto \mathbb{V}_2 : \_}{\Theta; \Gamma \vdash \mathrm{P} \preceq \Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \leadsto \lambda(\mathrm{X}{:}\mathrm{S}_1)\,\mathsf{:>}\mathrm{S}_2.(\mathsf{let}\ \mathrm{Z}{=}\mathrm{P}\langle \mathbb{V}_1 \rangle\ \mathsf{in}\ \mathbb{V}_2\ \mathsf{:>}\ \mathrm{S}_2) : \Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2}$$

$$\frac{\mathrm{S}_2 = \{\ldots\} \quad \Gamma \vdash \mathrm{P} : \Pi \mathrm{Y}{:}\mathrm{R}_1.\mathrm{R}_2 \quad \Theta; \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{R}_1 \leadsto \mathbb{V}_1 : \_ \quad \Theta; \Gamma, \mathrm{X}{:}\mathrm{S}_1, \mathrm{Z}{:}\mathrm{R}_2[\mathbb{V}_1/\mathrm{Y}] \vdash \mathrm{Z} \preceq \mathrm{S}_2 \leadsto \mathbb{V}_2 : \_}{\Theta; \Gamma \vdash \mathrm{P} \preceq \Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \leadsto \lambda(\mathrm{X}{:}\mathrm{S}_1)\,\mathsf{:>}\mathrm{S}_2.(\mathsf{let}\ \mathrm{Z}{=}\mathrm{P}(\mathbb{V}_1)\ \mathsf{in}\ \mathbb{V}_2\ \mathsf{:>}\ \mathrm{S}_2) : \Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2}$$

Figure 9. Elaboration Rules for Coercive Signature Matching

Top-Level Modules: $\Theta; \Gamma \vdash \mathit{top} \leadsto \mathrm{M} : \mathrm{S}$

Analogous to module elaboration rules, minus the rule for $[\mathit{exp}]$, but plus:

$$\frac{\Theta; \Gamma \vdash \mathrm{P}\ \mathsf{usable} \quad \Theta, \mathrm{P}; \Gamma \vdash \mathit{top} \leadsto \mathrm{M} : \mathrm{S}}{\Theta; \Gamma \vdash \mathsf{using}\ \mathrm{P}\ \mathsf{in}\ \mathit{top} \leadsto \mathrm{M} : \mathrm{S}}$$

Contexts: $\vdash (\Theta; \Gamma)\ \mathsf{ok}$

$$\frac{\vdash \Gamma\ \mathsf{ok}}{\vdash (\emptyset; \Gamma)\ \mathsf{ok}} \qquad \frac{\vdash (\Theta; \Gamma)\ \mathsf{ok} \quad \Theta; \Gamma \vdash \mathrm{P}\ \mathsf{usable}}{\vdash (\Theta, \mathrm{P}; \Gamma)\ \mathsf{ok}}$$

Figure 10. Elaboration Rules for Top-Level Modules and Well-formedness Rules for Contexts

Canonical Modules: $\Theta; \Gamma \models_{\mathrm{can}} \mathbb{V} : \mathbb{S}$

$$\frac{\Gamma \vdash \tau : \mathbf{T}}{\Theta; \Gamma \models_{\mathrm{can}} [\tau] : \llbracket \mathfrak{S}(\tau) \rrbracket} \qquad \frac{\Gamma, \alpha{:}\mathbf{T}^n \vdash \tau : \mathbf{T}}{\Theta; \Gamma \models_{\mathrm{can}} [\lambda \alpha{:}\mathbf{T}^n.\tau] : \llbracket \Pi \alpha{:}\mathbf{T}^n.\mathfrak{S}(\tau) \rrbracket} \qquad \frac{\forall i \in 1..n :\ \Theta; \Gamma \models_{\mathrm{can}} \mathbb{V}_i : \mathbb{S}_i \quad \mathtt{t} \notin \{\ell_1, \ldots, \ell_n\}}{\Theta; \Gamma \models_{\mathrm{can}} \{\ell_1{=}\mathbb{V}_1, \ldots, \ell_n{=}\mathbb{V}_n\} : \{\ell_1{:}\mathbb{S}_1, \ldots, \ell_n{:}\mathbb{S}_n\}}$$

$$\frac{\mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} : \mathbb{S} \quad \mathbb{S} = \{\mathtt{t}{:}\llbracket \mathfrak{S}(\tau) \rrbracket, \ldots\}}{\Theta; \Gamma \models_{\mathrm{can}} \mathrm{P} : \mathbb{S}} \qquad \frac{\mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} : \forall \mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \quad \Theta; \Gamma \models_{\mathrm{can}} \mathbb{V} : \mathbb{S}_1 \quad \Gamma \vdash \mathbb{S}_1 \leq \mathrm{S}_1}{\Theta; \Gamma \models_{\mathrm{can}} \mathrm{P}\langle \mathbb{V} \rangle : \mathbb{S}_2[\mathbb{V}/\mathrm{X}]} \qquad \frac{\Theta; \Gamma \models_{\mathrm{can}} \mathbb{V} : \mathbb{S}' \quad \Gamma \vdash \mathbb{S}' \equiv \mathbb{S}}{\Theta; \Gamma \models_{\mathrm{can}} \mathbb{V} : \mathbb{S}}$$

Figure 11. Elaboration Rules for Canonical Modules

Signature Similarity: $\mathrm{R} \approx \mathrm{S}$

$$\frac{\mathrm{R} = \{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{R}_1, \ldots, \ell_n \triangleright \mathrm{X}_n{:}\mathrm{R}_n\} \quad \mathrm{S} = \{\ell_1 \triangleright \mathrm{Y}_1{:}\mathrm{S}_1, \ldots, \ell_n \triangleright \mathrm{Y}_n{:}\mathrm{S}_n\}}{\mathrm{R} \approx \mathrm{S}}$$

Abstract Type Extraction: $\mathrm{S} \rightrightarrows \exists\overline{\alpha}.\mathbb{S}$

$$\frac{}{\mathbb{S} \rightrightarrows \exists\emptyset.\mathbb{S}} \qquad \frac{}{\llbracket \mathbf{T} \rrbracket \rightrightarrows \exists\alpha.\llbracket \boldsymbol{\mathfrak{S}}(\alpha) \rrbracket} \qquad \frac{\mathrm{S}_1 \rightrightarrows \exists\overline{\alpha_1}.\mathbb{S}_1 \quad \{\overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\} \rightrightarrows \exists\overline{\alpha}.\{\overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\} \quad \overline{\alpha_1} \cap \overline{\alpha} = \emptyset}{\{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\} \rightrightarrows \exists\overline{\alpha_1}, \overline{\alpha}.\{\ell_1 \triangleright \mathrm{X}_1{:}\mathbb{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}}$$

Class Parameters: $\mathsf{params}(\mathbb{S})$

$$\begin{array}{rcl}
\mathsf{params}(\llbracket \boldsymbol{\mathfrak{S}}(\alpha) \rrbracket) & \overset{\text{def}}{=} & \{\alpha\} \\
\mathsf{params}(\{\}) & \overset{\text{def}}{=} & \emptyset \\
\mathsf{params}(\{\ell_1 \triangleright \mathrm{X}_1{:}\mathbb{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}) & \overset{\text{def}}{=} & \mathsf{params}(\mathbb{S}_1) \cup \mathsf{params}(\{\overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}),\ \text{where } \mathtt{t} \notin \{\ell_1, \overline{\ell}\} \\
\mathsf{params}(\{\mathtt{t} \triangleright \mathrm{X}{:}\llbracket \boldsymbol{\mathfrak{S}}(\alpha) \rrbracket, \ldots\}) & \overset{\text{def}}{=} & \{\alpha\}
\end{array}$$

Paths to Atomic Instance Modules in a Class: $\mathsf{paths}(\mathrm{P} : \mathbb{S})$

$$\begin{array}{rcl}
\mathsf{paths}(\mathrm{P} : \llbracket \boldsymbol{\mathfrak{S}}(\tau) \rrbracket) & \overset{\text{def}}{=} & \emptyset \\
\mathsf{paths}(\mathrm{P} : \{\}) & \overset{\text{def}}{=} & \emptyset \\
\mathsf{paths}(\mathrm{P} : \{\ell_1 \triangleright \mathrm{X}_1{:}\mathbb{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}) & \overset{\text{def}}{=} & \mathsf{paths}(\mathrm{P}.\ell_1 : \mathbb{S}_1) \cup \mathsf{paths}(\mathrm{P} : \{\overline{\ell \triangleright \mathrm{X}{:}\mathbb{S}}\}),\ \text{where } \mathtt{t} \notin \{\ell_1, \overline{\ell}\} \\
\mathsf{paths}(\mathrm{P} : \{\mathtt{t} \triangleright \mathrm{X}{:}\llbracket \boldsymbol{\mathfrak{S}}(\tau) \rrbracket, \ldots\}) & \overset{\text{def}}{=} & \{\mathrm{P}\}
\end{array}$$

Instance Set Simplification: $\Gamma \vdash \Theta \downarrow \Theta'$

$$\frac{\begin{array}{c}\Theta' \subseteq \Theta \\ \forall \mathrm{P}_1 \in \Theta.\ \exists \mathrm{P}_2 \in \Theta'.\ \Gamma \vdash \mathrm{P}_1 : \mathbb{S}_1 \wedge \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \wedge \Gamma \vdash \mathbb{S}_1 \equiv \mathbb{S}_2 \\ \forall \mathrm{P}_1, \mathrm{P}_2 \in \Theta'.\ (\mathrm{P}_1 \neq \mathrm{P}_2 \wedge \Gamma \vdash \mathrm{P}_1 : \mathbb{S}_1 \wedge \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \wedge \mathbb{S}_1 \approx \mathbb{S}_2) \Rightarrow \Gamma \vdash \mathrm{P}_1.\mathtt{t} \not\equiv \mathrm{P}_2.\mathtt{t} : \mathbf{T}\end{array}}{\Gamma \vdash \Theta \downarrow \Theta'}$$

Class Elaboration: $\Gamma \vdash_{\text{class}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta$

$$\frac{\Gamma \vdash \mathrm{S}\ \text{sig} \quad \mathrm{S} \rightrightarrows \exists\overline{\alpha}.\mathbb{S} \quad \Gamma, \overline{\alpha} \vdash \mathbb{S} \downarrow \mathbb{S}' \quad \mathsf{params}(\mathbb{S}') \subseteq \{\overline{\alpha}\} \quad \Gamma, \overline{\alpha}, \mathrm{X}{:}\mathbb{S} \vdash \mathsf{paths}(\mathrm{X} : \mathbb{S}) \downarrow \Theta}{\Gamma \vdash_{\text{class}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta}$$

Well-formed Classes: $\Gamma \vdash \mathrm{S}\ \mathsf{class}$

$$\frac{\Gamma \vdash_{\text{class}} \mathrm{X} : \mathrm{S} \rightsquigarrow \Theta}{\Gamma \vdash \mathrm{S}\ \mathsf{class}}$$

Figure 12. Elaboration Rules for Classes

Concrete Signatures: $\Gamma \vdash \mathrm{S}$ concrete

$$\frac{\Gamma \vdash \tau : \mathbf{T}}{\Gamma \vdash [\![\mathfrak{S}(\tau)]\!] \text{ concrete}} \qquad \frac{\forall \mathrm{S} \in \overline{\mathrm{S}}.\ \Gamma \vdash \mathrm{S} \text{ concrete} \quad \mathtt{t} \notin \overline{\ell}}{\Gamma \vdash \{\overline{\ell \triangleright \mathrm{X{:}S}}\} \text{ concrete}} \qquad \frac{\Gamma \vdash \tau : \mathbf{T}}{\Gamma \vdash \{\mathtt{t} \triangleright \mathrm{X}{:}[\![\mathfrak{S}(\tau)]\!], \ldots\} \text{ concrete}} \qquad \frac{\Gamma \vdash \mathrm{S}' \text{ concrete} \quad \Gamma \vdash \mathrm{S}' \equiv \mathrm{S}}{\Gamma \vdash \mathrm{S} \text{ concrete}}$$

Instances: $\Gamma \vdash (\mathrm{P}; p)$ instance

$$\frac{\Gamma \vdash \mathrm{P} : \mathbb{S} \quad \mathbb{S} = \{\mathtt{t} \triangleright \mathrm{Y}{:}[\![\mathfrak{S}(\mathrm{C})]\!], \ldots\} \quad \Gamma \vdash \mathrm{C} \downarrow p}{\Gamma \vdash (\mathrm{P}; p) \text{ instance}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{P} : \forall \mathrm{X{:}S}_1.\mathbb{S}_2 \quad \Gamma \vdash \mathrm{S}_1 \text{ class} \quad \mathrm{S}_1 \rightrightarrows \exists \overline{\alpha}.\mathbb{S}_1 \quad \mathbb{S}_2 = \{\mathtt{t} \triangleright \mathrm{Y}{:}[\![\mathfrak{S}(\mathrm{C})]\!], \ldots\} \\ \Gamma, \overline{\alpha} \vdash \mathbb{S}_1 \downarrow \mathbb{S}_1' \quad \text{params}(\mathbb{S}_1') = \{\overline{\beta}\} \quad \Gamma, \overline{\alpha}, \mathrm{X}{:}\mathbb{S}_1 \vdash \mathrm{C} \downarrow p(\overline{\beta})\end{array}}{\Gamma \vdash (\mathrm{P}; p) \text{ instance}}$$

Usable Instances: $\Theta; \Gamma \vdash \mathrm{P}$ usable

$$\frac{\Gamma \vdash (\mathrm{P}; p) \text{ instance} \quad \Gamma \vdash \mathrm{P} : \mathbb{S} \quad \forall \mathrm{P}' \in \Theta.\ (\Gamma \vdash (\mathrm{P}'; p) \text{ instance} \wedge \Gamma \vdash \mathrm{P}' : \mathbb{S}') \Rightarrow \mathbb{S} \not\approx \mathbb{S}'}{\Theta; \Gamma \vdash \mathrm{P} \text{ usable}}$$

$$\frac{\Gamma \vdash (\mathrm{P}; p) \text{ instance} \quad \Gamma \vdash \mathrm{P} : \forall \mathrm{X{:}S}_1.\mathbb{S} \quad \forall \mathrm{P}' \in \Theta.\ (\Gamma \vdash (\mathrm{P}'; p) \text{ instance} \wedge \Gamma \vdash \mathrm{P}' : \forall \mathrm{X{:}S}_1'.\mathbb{S}') \Rightarrow \mathbb{S} \not\approx \mathbb{S}'}{\Theta; \Gamma \vdash \mathrm{P} \text{ usable}}$$

Figure 13. Elaboration Rules for Instances

$\boxed{\textbf{Type Unification:}\ \Gamma \vdash \tau_1 = \tau_2 \Rightarrow \delta}$

$$\frac{}{\Gamma \vdash \tau = \tau \Rightarrow \mathbf{id}} \qquad \frac{\Gamma \vdash \tau_1 = \tau_1' \Rightarrow \delta_1 \quad \delta_1\Gamma \vdash \delta_1\tau_2 = \delta_1\tau_2' \Rightarrow \delta_2}{\Gamma \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' \Rightarrow \delta_2\delta_1} \qquad \frac{\Gamma \vdash \overline{\tau_1} = \overline{\tau_2} \Rightarrow \delta}{\Gamma \vdash p(\overline{\tau_1}) = p(\overline{\tau_2}) \Rightarrow \delta}$$

$$\frac{\boldsymbol{\alpha} \notin \mathrm{FV}(\tau) \quad \vdash \Gamma[\tau/\boldsymbol{\alpha}]\ \mathsf{ok}}{\Gamma \vdash \boldsymbol{\alpha} = \tau \Rightarrow \{\boldsymbol{\alpha} \mapsto \tau\}} \qquad \frac{\boldsymbol{\alpha} \notin \mathrm{FV}(\tau) \quad \vdash \Gamma[\tau/\boldsymbol{\alpha}]\ \mathsf{ok}}{\Gamma \vdash \tau = \boldsymbol{\alpha} \Rightarrow \{\boldsymbol{\alpha} \mapsto \tau\}}$$

$\boxed{\textbf{Kind Unification:}\ \Gamma \vdash \mathrm{K}_1 = \mathrm{K}_2 \Rightarrow \delta}$

$$\frac{}{\Gamma \vdash \mathrm{K} = \mathrm{K} \Rightarrow \mathbf{id}} \qquad \frac{\Gamma \vdash \tau_1 = \tau_2 \Rightarrow \delta}{\Gamma \vdash \mathfrak{S}(\tau_1) = \mathfrak{S}(\tau_2) \Rightarrow \delta} \qquad \frac{\Gamma, \alpha{:}\mathbf{T}^n \vdash \tau_1 = \tau_2 \Rightarrow \delta \quad \alpha \notin \mathrm{FV}(\delta)}{\Gamma \vdash \Pi\alpha{:}\mathbf{T}^n.\mathfrak{S}(\tau_1) = \Pi\alpha{:}\mathbf{T}^n.\mathfrak{S}(\tau_2) \Rightarrow \delta}$$

$\boxed{\textbf{Signature Unification:}\ \Gamma \vdash \mathrm{S}_1 = \mathrm{S}_2 \Rightarrow \delta}$

$$\frac{\Gamma \vdash \mathrm{K}_1 = \mathrm{K}_2 \Rightarrow \delta}{\Gamma \vdash \llbracket \mathrm{K}_1 \rrbracket = \llbracket \mathrm{K}_2 \rrbracket \Rightarrow \delta} \qquad \frac{\Gamma \vdash \tau_1 = \tau_2 \Rightarrow \delta}{\Gamma \vdash \llbracket \tau_1 \rrbracket = \llbracket \tau_2 \rrbracket \Rightarrow \delta}$$

$$\frac{}{\Gamma \vdash \{\} = \{\} \Rightarrow \mathbf{id}} \qquad \frac{\Gamma \vdash \mathrm{S}_1 = \mathrm{S}_1' \Rightarrow \delta_1 \quad \delta_1\Gamma, \mathrm{X}_1{:}\delta_1\mathrm{S}_1 \vdash \delta_1\{\overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\} = \delta_1\{\overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}'}\} \Rightarrow \delta_2 \quad \mathrm{X}_1^{\mathsf{c}} \notin \mathrm{FV}(\delta_2)}{\Gamma \vdash \{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}}\} = \{\ell_1 \triangleright \mathrm{X}_1{:}\mathrm{S}_1', \overline{\ell \triangleright \mathrm{X}{:}\mathrm{S}'}\} \Rightarrow \delta_2\delta_1}$$

$$\frac{\Gamma \vdash \mathrm{S}_1 = \mathrm{S}_1' \Rightarrow \delta_1 \quad \delta_1\Gamma, \mathrm{X}{:}\delta_1\mathrm{S}_1 \vdash \delta_1\mathrm{S}_2 = \delta_1\mathrm{S}_2' \Rightarrow \delta_2 \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\delta_2)}{\Gamma \vdash \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 = \Pi\mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_2' \Rightarrow \delta_2\delta_1}$$

$$\frac{\Gamma \vdash \mathrm{S}_1 = \mathrm{S}_1' \Rightarrow \delta_1 \quad \delta_1\Gamma, \mathrm{X}{:}\delta_1\mathrm{S}_1 \vdash \delta_1\mathrm{S}_2 = \delta_1\mathrm{S}_2' \Rightarrow \delta_2 \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\delta_2)}{\Gamma \vdash \forall\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 = \forall\mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_2' \Rightarrow \delta_2\delta_1}$$

$\boxed{\textbf{Multiple Unifications:}\ \Gamma \vdash \overline{meta_1} = \overline{meta_2} \Rightarrow \delta}$

$$\frac{}{\Gamma \vdash \emptyset = \emptyset \Rightarrow \mathbf{id}} \qquad \frac{\Gamma \vdash meta_1 = meta_1' \Rightarrow \delta_1 \quad \delta_1\Gamma \vdash \delta_1\overline{meta} = \delta_1\overline{meta'} \Rightarrow \delta_2}{\Gamma \vdash meta_1, \overline{meta} = meta_1', \overline{meta'} \Rightarrow \delta_2\delta_1}$$

$\boxed{\textbf{Unification After Normalization:}\ \Gamma \vdash meta_1 \equiv meta_2 \Rightarrow \delta}$

$$\frac{\Gamma \vdash meta_1 \downarrow meta_1' \quad \Gamma \vdash meta_2 \downarrow meta_2' \quad \Gamma \vdash meta_1' = meta_2' \Rightarrow \delta}{\Gamma \vdash meta_1 \equiv meta_2 \Rightarrow \delta}$$

Figure 14. Unification Rules

$\boxed{\textbf{Kinds:}\ \Theta; \Gamma \vdash knd \Rightarrow \mathrm{K}}$

$\boxed{\textbf{Signatures:}\ \Theta; \Gamma \vdash sig \Rightarrow \mathrm{S}}$

$\boxed{\textbf{Constructors:}\ \Theta; \Gamma \vdash con \Rightarrow \mathrm{C} : \mathrm{K}}$

Analogous to corresponding declarative elaboration rules, except:

$$\frac{\Theta; \Gamma \vdash \mathsf{canon}(sig) \Rightarrow_{\downarrow} \mathbb{V} : \mathbb{S}/(\emptyset; \mathbf{id}) \quad \Gamma \vdash \mathbb{V}.\ell s : \llbracket \mathrm{K} \rrbracket}{\Theta; \Gamma \vdash \mathsf{canon}(sig).\ell s \Rightarrow \mathsf{Fst}(\mathbb{V}).\ell s : \mathrm{K}}$$

Figure 15. Type Inference Rules for Kinds, Constructors and Signatures

Terms With Constraint and Type Normalization: $\Theta; \Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma; \delta)$

$$\frac{\Theta; \Gamma \vdash \mathit{exp} \Rightarrow e : \tau'/(\Sigma_1; \delta_1) \quad \Theta; \delta_1\Gamma \vdash \Sigma_1 \downarrow (\Sigma_2; \sigma; \delta_2) \quad \delta_2\delta_1\Gamma \vdash \delta_2\tau' \downarrow \tau}{\Theta; \Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow \sigma\delta_2 e : \tau/(\Sigma_2; \delta_2\delta_1|_\Gamma)}$$

Terms With Constraint Normalization Given a Ground Target Type: $\Theta; \Gamma \vdash \mathit{exp} : \tau \Rightarrow\!\downarrow e/(\Sigma; \delta)$

$$\frac{\Theta; \Gamma \vdash \mathit{exp} \Rightarrow e : \tau'/(\Sigma_1; \delta_1) \quad \delta_1\Gamma \vdash \tau' \equiv \tau \Rightarrow \delta_2 \quad \Theta; \delta_2\delta_1\Gamma \vdash \delta_2\Sigma_1 \downarrow (\Sigma_2; \sigma; \delta_3)}{\Theta; \Gamma \vdash \mathit{exp} : \tau \Rightarrow\!\downarrow \sigma\delta_3\delta_2 e/(\Sigma_2; \delta_3\delta_2\delta_1|_\Gamma)}$$

Terms: $\Theta; \Gamma \vdash \mathit{exp} \Rightarrow e : \tau/(\Sigma; \delta)$

$$\frac{x{:}\tau \in \Gamma}{\Theta; \Gamma \vdash x \Rightarrow x : \tau/(\emptyset; \mathbf{id})} \qquad \frac{\Theta; \Gamma, x{:}\boldsymbol{\alpha} \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma; \delta)}{\Theta; \Gamma \vdash \lambda x.\mathit{exp} \Rightarrow \lambda x{:}\delta\boldsymbol{\alpha}.e : \delta\boldsymbol{\alpha} \to \tau/(\Sigma; \delta|_\Gamma)}$$

$$\frac{\Theta; \Gamma \vdash \mathit{exp}_1 \Rightarrow\!\downarrow e_1 : \tau_1/(\Sigma_1; \delta_1) \quad \Theta; \delta_1\Gamma \vdash \mathit{exp}_2 \Rightarrow\!\downarrow e_2 : \tau_2/(\Sigma_2; \delta_2) \quad \delta_2\delta_1\Gamma \vdash \delta_2\tau_1 \equiv (\tau_2 \to \boldsymbol{\alpha}) \Rightarrow \delta_3}{\Theta; \Gamma \vdash \mathit{exp}_1(\mathit{exp}_2) \Rightarrow \delta_3\delta_2 e_1(\delta_3 e_2) : \delta_3\boldsymbol{\alpha}/(\delta_3\delta_2\Sigma_1, \delta_3\Sigma_2; \delta_3\delta_2\delta_1|_\Gamma)}$$

$$\frac{\Gamma \vdash \mathrm{P} :\downarrow \llbracket\tau\rrbracket}{\Theta; \Gamma \vdash \mathrm{P} \Rightarrow \mathsf{Val}(\mathrm{P}) : \tau/(\emptyset; \mathbf{id})} \qquad \frac{\Gamma \vdash \mathrm{P} :\downarrow \forall \mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket \quad \mathrm{S} \rightrightarrows \exists\overline{\boldsymbol{\alpha}}.\mathbb{S} \quad \Gamma, \mathrm{X}{:}\mathbb{S} \vdash \tau \downarrow \tau'}{\Theta; \Gamma \vdash \mathrm{P} \Rightarrow \mathsf{Val}(\mathrm{P}\langle\mathrm{X}\rangle) : \tau'/(\mathrm{X}{:}\mathbb{S}; \mathbf{id})}$$

$$\frac{\Theta; \Gamma \vdash \mathit{mod} \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}/(\Sigma_1; \delta_1) \quad \Theta; \delta_1\Gamma, \mathrm{X}{:}\mathrm{S} \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma_2; \delta_2) \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\tau, \Sigma_2)}{\Theta; \Gamma \vdash \mathsf{let}\,\mathrm{X}{=}\mathit{mod}\,\mathsf{in}\;\mathit{exp} \Rightarrow \mathsf{let}\,\mathrm{X}{=}\delta_2\mathrm{M}\,\mathsf{in}\;e : \tau/(\delta_2\Sigma_1, \Sigma_2; \delta_2\delta_1|_\Gamma)}$$

$$\frac{\Theta; \Gamma \vdash \mathit{typ} \Rightarrow \tau : \mathbf{T} \quad \Theta; \Gamma \vdash \mathit{exp} : \tau \Rightarrow\!\downarrow e/(\Sigma; \delta)}{\Theta; \Gamma \vdash \mathit{exp} : \mathit{typ} \Rightarrow e : \tau/(\Sigma; \delta)}$$

Figure 16. Type Inference Rules for Terms

$$\begin{array}{rcl}
\mathsf{partition}(\overline{\boldsymbol{\alpha}}; \Sigma) & \overset{\text{def}}{=} & (\Sigma_1; \Sigma_2), \\
& & \text{where } \Sigma_2 = \{\mathrm{X}{:}\mathbb{S} \mid \mathrm{X}{:}\mathbb{S} \in \Sigma \wedge \exists\boldsymbol{\alpha} \in \overline{\boldsymbol{\alpha}}.\ \mathbb{S} = \{\mathtt{t}{:}\boldsymbol{\mathfrak{S}}(\boldsymbol{\alpha}), \ldots\}\} \text{ and } \Sigma_1 = \Sigma - \Sigma_2 \\
\mathsf{makesig}(\overline{\boldsymbol{\alpha}}; \Sigma) & \overset{\text{def}}{=} & \{\mathsf{tyvars}\triangleright \mathrm{Y}{:}\{1{:}\llbracket\mathbf{T}\rrbracket, \ldots, m{:}\llbracket\mathbf{T}\rrbracket\}, \mathsf{consts}{:}\{1{:}\mathbb{S}'_1, \ldots, n{:}\mathbb{S}'_n\}\}, \\
& & \text{where } \overline{\boldsymbol{\alpha}} = \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m \text{ and } \Sigma = \mathrm{X}_1{:}\mathbb{S}_1, \ldots, \mathrm{X}_n{:}\mathbb{S}_n \text{ and } \mathbb{S}'_i = \mathbb{S}_i[\mathrm{Y}.j/\boldsymbol{\alpha}_j]_{j=1}^n \\
\mathsf{genvars}(\Gamma; \Sigma; \tau) & \overset{\text{def}}{=} & \overline{\boldsymbol{\alpha}}_2, \\
& & \text{where } \overline{\boldsymbol{\alpha}}_2 \text{ is the greatest set such that } \overline{\boldsymbol{\alpha}}_1 \cup \overline{\boldsymbol{\alpha}}_2 = \mathrm{UV}(\Sigma, \tau) \text{ and } \overline{\boldsymbol{\alpha}}_1 \cap \overline{\boldsymbol{\alpha}}_2 = \emptyset \\
& & \text{and } \overline{\boldsymbol{\alpha}}_2 \cap \mathrm{UV}(\Gamma, \Sigma_1) = \emptyset\text{, where } \mathsf{partition}(\overline{\boldsymbol{\alpha}}_2; \Sigma) = (\Sigma_1; \Sigma_2)
\end{array}$$

Figure 17. Auxiliary Definitions for Use in the Polymorphic Generalization Rule

Modules With Constraint and Signature Normalization: $\Theta;\Gamma \vdash \mathit{mod} \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}/(\Sigma;\delta)$

$$\frac{\Theta;\Gamma \vdash \mathit{mod} \Rightarrow \mathrm{M} : \mathrm{S}/(\Sigma_1;\delta_1) \quad \Theta;\delta_1\Gamma \vdash \Sigma_1 \downarrow (\Sigma_2;\sigma;\delta_2) \quad \delta_2\delta_1\Gamma \vdash \delta_2\mathrm{S} \downarrow \mathrm{S}'}{\Theta;\Gamma \vdash \mathit{mod} \Rightarrow\!\downarrow \sigma\delta_2\mathrm{M} : \mathrm{S}'/(\Sigma_2;\delta_2\delta_1|_\Gamma)}$$

Modules: $\Theta;\Gamma \vdash \mathit{mod} \Rightarrow \mathrm{M} : \mathrm{S}/(\Sigma;\delta)$

$$\frac{\Gamma \vdash \mathrm{P} :\downarrow \mathbb{S}}{\Theta;\Gamma \vdash \mathrm{P} \Rightarrow \mathrm{P} : \mathbb{S}/(\emptyset;\mathbf{id})} \qquad \frac{\Theta;\Gamma \vdash \mathit{con} \Rightarrow \mathrm{C} : \mathbb{K}}{\Theta;\Gamma \vdash [\mathit{con}] \Rightarrow [\mathrm{C}] : \llbracket\mathbb{K}\rrbracket/(\emptyset;\mathbf{id})} \qquad \frac{\Theta;\Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma;\delta) \quad e \text{ not valuable}}{\Theta;\Gamma \vdash [\mathit{exp}] \Rightarrow [e] : \llbracket\tau\rrbracket/(\Sigma;\delta)}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow v : \tau/(\Sigma;\delta_1) \quad \mathsf{genvars}(\delta_1\Gamma;\Sigma;\tau) = \overline{\boldsymbol{\alpha}} \quad \mathsf{partition}(\overline{\boldsymbol{\alpha}};\Sigma) = (\Sigma_1;\Sigma_2) \quad \mathsf{makesig}(\overline{\boldsymbol{\alpha}};\Sigma_2) = \mathrm{S} \\ \overline{\boldsymbol{\alpha}} = \boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_m \quad \Sigma_2 = \mathrm{X}_1{:}\mathbb{S}_1,\ldots,\mathrm{X}_n{:}\mathbb{S}_n \quad \delta = \{\boldsymbol{\alpha}_i \mapsto \mathrm{X}.\mathsf{tyvars}.i\}_{i=1}^m \quad \sigma = \{\mathrm{X}_i \mapsto \mathrm{X}.\mathsf{consts}.i\}_{i=1}^n\end{array}}{\Theta;\Gamma \vdash [\mathit{exp}] \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.[\sigma\delta v] : \forall\mathrm{X}{:}\mathrm{S}.\llbracket\delta\tau\rrbracket/(\Sigma_1;\delta_1)}$$

$$\frac{}{\Theta;\Gamma \vdash \{\} \Rightarrow \{\} : \{\}/(\emptyset;\mathbf{id})}$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod}_1 \Rightarrow\!\downarrow \mathrm{M}_1 : \mathrm{S}_1/(\Sigma_1;\delta_1) \quad \Theta;\delta_1\Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \{\overline{\ell{\triangleright}\mathrm{X}{=}\mathit{mod}}\} \Rightarrow\!\downarrow \{\overline{\ell{\triangleright}\mathrm{X}{=}\mathrm{M}}\} : \{\overline{\ell{\triangleright}\mathrm{X}{:}\mathrm{S}}\}/(\Sigma_2;\delta_2) \quad \mathrm{X}_1^{\mathsf{c}} \notin \mathrm{FV}(\Sigma_2)}{\Theta;\Gamma \vdash \{\ell_1{\triangleright}\mathrm{X}_1{=}\mathit{mod}_1, \overline{\ell{\triangleright}\mathrm{X}{=}\mathit{mod}}\} \Rightarrow \{\ell_1{\triangleright}\mathrm{X}_1{=}\delta_2\mathrm{M}_1, \overline{\ell{\triangleright}\mathrm{X}{=}\mathrm{M}}\} : \{\ell_1{\triangleright}\mathrm{X}_1{:}\delta_2\mathrm{S}_1, \overline{\ell{\triangleright}\mathrm{X}{:}\mathrm{S}}\}/(\delta_2\Sigma_1,\Sigma_2;\delta_2\delta_1|_\Gamma)}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod} \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}_2/(\Sigma;\delta) \quad \mathrm{S}_2 = \{\ldots\} \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\Sigma)}{\Theta;\Gamma \vdash \lambda\mathrm{X}{:}\mathit{sig}.\mathit{mod} \Rightarrow \lambda(\mathrm{X}{:}\mathrm{S}_1)\mathrel{:>}\mathrm{S}_2.\mathrm{M} : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2/(\Sigma;\delta|_\Gamma)}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S}_1 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod} \Rightarrow\!\downarrow \mathrm{V} : \mathbb{S}_2/(\Sigma;\delta) \quad \mathbb{S}_2 = \{\ldots\} \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\Sigma)}{\Theta;\Gamma \vdash \Lambda\mathrm{X}{:}\mathit{sig}.\mathit{mod} \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}_1.\mathrm{V} : \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2/(\Sigma;\delta|_\Gamma)}$$

$$\frac{\Gamma \vdash \mathrm{P}_1 :\downarrow \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S} \quad \mathrm{S} = \{\ldots\} \quad \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \quad \Theta;\Gamma \vdash \mathrm{P}_2 \preceq \mathrm{S}_1 \Rightarrow \mathbb{V} : \mathbb{S}_1/\delta}{\Theta;\Gamma \vdash \mathrm{P}_1(\mathrm{P}_2) \Rightarrow \mathrm{P}_1(\mathbb{V}) : \delta\mathrm{S}[\mathbb{V}/\mathrm{X}]/(\emptyset;\delta)}$$

$$\frac{\Gamma \vdash \mathrm{P}_1 :\downarrow \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S} \quad \mathbb{S} = \{\ldots\} \quad \Gamma \vdash \mathrm{P}_2 : \mathbb{S}_2 \quad \Theta;\Gamma \vdash \mathrm{P}_2 \preceq \mathrm{S}_1 \Rightarrow \mathbb{V} : \mathbb{S}_1/\delta}{\Theta;\Gamma \vdash \mathrm{P}_1\langle\mathrm{P}_2\rangle \Rightarrow \mathrm{P}_1\langle\mathbb{V}\rangle : \delta\mathbb{S}[\mathbb{V}/\mathrm{X}]/(\emptyset;\delta)}$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod}_1 \Rightarrow\!\downarrow \mathrm{M}_1 : \mathrm{S}_1/(\Sigma_1;\delta_1) \quad \Theta;\delta_1\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathit{mod}_2 \Rightarrow\!\downarrow \mathrm{M}_2 : \mathrm{S}/(\Sigma_2;\delta_2) \quad \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathrm{S},\Sigma_2)}{\Theta;\Gamma \vdash \mathsf{let}\,\mathrm{X}{=}\mathit{mod}_1\,\mathsf{in}\,\mathit{mod}_2 \Rightarrow \mathsf{let}\,\mathrm{X}{=}\delta_2\mathrm{M}_1\,\mathsf{in}\,\mathrm{M}_2 \mathrel{:>} \mathrm{S} : \mathrm{S}/(\delta_2\Sigma_1,\Sigma_2;\delta_2\delta_1|_\Gamma)}$$

$$\frac{\Theta;\Gamma \vdash \mathit{mod} \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}_1/(\Sigma;\delta_1) \quad \Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S} \quad \Theta;\delta_1\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{S} \Rightarrow \mathbb{V} : \mathbb{S}/\delta_2}{\Theta;\Gamma \vdash \mathit{mod} \mathrel{:>} \mathit{sig} \Rightarrow \mathsf{let}\,\mathrm{X}{=}\delta_2\mathrm{M}\,\mathsf{in}\,\mathbb{V} \mathrel{:>} \mathrm{S} : \mathrm{S}/(\delta_2\Sigma;\delta_2\delta_1|_\Gamma)}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S} \quad \Gamma \vdash \mathrm{S}\ \mathsf{concrete} \quad \mathrm{S} \rightrightarrows \exists\overline{\boldsymbol{\alpha}}.\mathbb{S} \quad \Theta;\Gamma \vdash \mathrm{X}{:}\mathbb{S} \downarrow (\emptyset;\sigma;\delta) \quad \overline{\boldsymbol{\alpha}} \subseteq \mathrm{dom}(\delta)}{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \Rightarrow \sigma\mathrm{X} : \delta\mathbb{S}/(\emptyset;\mathbf{id})}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S} \quad \Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathrm{X}.\ell s :\downarrow \llbracket\tau\rrbracket \quad \Gamma \vdash \mathrm{S}\ \mathsf{class}}{\Theta;\Gamma \vdash \mathsf{overload}\ \ell s\ \mathsf{from}\ \mathit{sig} \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{X}.\ell s) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket/(\emptyset;\mathbf{id})}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S}_1 \quad \Gamma,\mathrm{X}_1{:}\mathrm{S}_1 \vdash \mathrm{X}_1.\ell s :\downarrow \forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket\tau\rrbracket \quad \mathrm{S} = \{1{\triangleright}\mathrm{X}_1{:}\mathrm{S}_1, 2{\triangleright}\mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\ \mathsf{class}}{\Theta;\Gamma \vdash \mathsf{overload}\ \ell s\ \mathsf{from}\ \mathit{sig} \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{X}.1.\ell s\langle\mathrm{X}.2\rangle) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau[\mathrm{X}.i/\mathrm{X}_i]\rrbracket/(\emptyset;\mathbf{id})}$$

$$\frac{\Gamma \vdash \mathrm{P} :\downarrow \forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket\tau\rrbracket\} \quad \Gamma \vdash \mathrm{S}\ \mathsf{class}}{\Theta;\Gamma \vdash \mathsf{implicit}(\mathrm{P}) \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{P}\langle\mathrm{X}\rangle.\mathsf{it}) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket/(\emptyset;\mathbf{id})}$$

$$\frac{\Gamma \vdash \mathrm{P} :\downarrow \forall\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{:}\forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket\tau\rrbracket\} \quad \mathrm{S} = \{1{\triangleright}\mathrm{X}_1{:}\mathrm{S}_1, 2{\triangleright}\mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\ \mathsf{class}}{\Theta;\Gamma \vdash \mathsf{implicit}(\mathrm{P}) \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.(\mathrm{P}\langle\mathrm{X}.1\rangle.\mathsf{it}\langle\mathrm{X}.2\rangle) : \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau[\mathrm{X}.i/\mathrm{X}_i]\rrbracket/(\emptyset;\mathbf{id})}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket\tau\rrbracket\} \quad \Gamma \vdash \mathrm{S}\ \mathsf{class} \quad \Theta;\Gamma \vdash \mathrm{P} \preceq \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket \Rightarrow \mathbb{V} : \_\,/\delta}{\Theta;\Gamma \vdash \mathsf{explicit}(\mathrm{P} : \mathit{sig}) \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{=}\mathbb{V}\langle\mathrm{X}\rangle\} : \forall\mathrm{X}{:}\mathrm{S}.\{\mathsf{it}{:}\llbracket\tau\rrbracket\}/(\emptyset;\delta)}$$

$$\frac{\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \forall\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{:}\forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket\tau\rrbracket\} \quad \mathrm{S} = \{1{\triangleright}\mathrm{X}_1{:}\mathrm{S}_1, 2{\triangleright}\mathrm{X}_2{:}\mathrm{S}_2\} \quad \Gamma \vdash \mathrm{S}\ \mathsf{class} \quad \Theta;\Gamma \vdash \mathrm{P} \preceq \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau[\mathrm{X}.i/\mathrm{X}_i]\rrbracket \Rightarrow \mathbb{V} : \_\,/\delta}{\Theta;\Gamma \vdash \mathsf{explicit}(\mathrm{P} : \mathit{sig}) \Rightarrow \Lambda\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{=}\Lambda\mathrm{X}_2{:}\mathrm{S}_2.\mathbb{V}\langle\{1{=}\mathrm{X}_1, 2{=}\mathrm{X}_2\}\rangle\} : \forall\mathrm{X}_1{:}\mathrm{S}_1.\{\mathsf{it}{:}\forall\mathrm{X}_2{:}\mathrm{S}_2.\llbracket\tau\rrbracket\}/(\emptyset;\delta)}$$

Figure 18. Type Inference Rules for Modules

Coercive Signature Matching: $\Theta;\Gamma \vdash \mathrm{P} \preceq \mathrm{S} \Rightarrow \mathbb{V} : \mathbb{S}/\delta$

$$\frac{\Gamma \vdash \mathrm{P} :\downarrow \llbracket\mathbb{K}\rrbracket \quad \Gamma \vdash \mathbb{K} \le \mathrm{K}}{\Theta;\Gamma \vdash \mathrm{P} \preceq \llbracket\mathrm{K}\rrbracket \Rightarrow \mathrm{P} : \llbracket\mathbb{K}\rrbracket/\mathbf{id}} \qquad \frac{\Theta;\Gamma \vdash \mathrm{P} : \tau \Rightarrow\!\downarrow v/(\emptyset;\delta)}{\Theta;\Gamma \vdash \mathrm{P} \preceq \llbracket\tau\rrbracket \Rightarrow [v] : \llbracket\tau\rrbracket/\delta} \qquad \frac{\Gamma \vdash_{\mathsf{class}} \mathrm{X} : \mathrm{S} \leadsto \Theta' \quad \Theta,\Theta';\Gamma,\mathrm{X}{:}\mathrm{S} \vdash \mathrm{P} : \tau \Rightarrow\!\downarrow v/(\emptyset;\delta)}{\Theta;\Gamma \vdash \mathrm{P} \preceq \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}.[v] : \forall\mathrm{X}{:}\mathrm{S}.\llbracket\tau\rrbracket/\delta}$$

$$\frac{\Gamma \vdash \mathrm{P} : \{\ldots\}}{\Theta;\Gamma \vdash \mathrm{P} \preceq \{\} \Rightarrow \{\} : \{\}/\mathbf{id}}$$

$$\frac{\Gamma \vdash \mathrm{P}.\ell_1 : \mathrm{S}_1' \quad \Theta;\Gamma \vdash \mathrm{P}.\ell_1 \preceq \mathrm{S}_1 \Rightarrow \mathbb{V}_1 : \mathbb{S}_1/\delta_1 \quad \Theta;\delta_1\Gamma,\mathrm{X}_1{:}\mathbb{S}_1 \vdash \mathrm{P} \preceq \{\overline{\ell \rhd \mathrm{X}{:}\mathrm{S}}\} \Rightarrow \{\overline{\ell \rhd \mathrm{X}{=}\mathbb{V}}\} : \{\overline{\ell \rhd \mathrm{X}{:}\mathbb{S}}\}/\delta_2}{\Theta;\Gamma \vdash \mathrm{P} \preceq \{\ell_1 \rhd \mathrm{X}_1{:}\mathrm{S}_1, \overline{\ell \rhd \mathrm{X}{:}\mathrm{S}}\} \Rightarrow \{\ell_1 \rhd \mathrm{X}_1{=}\delta_2\mathbb{V}_1, \overline{\ell \rhd \mathrm{X}{=}\mathbb{V}}\} : \{\ell_1 \rhd \mathrm{X}_1{:}\mathbb{S}_1, \overline{\ell \rhd \mathrm{X}{:}\mathbb{S}}\}/\delta_2\delta_1|_\Gamma}$$

$$\frac{\mathbb{S}_2 = \{\ldots\} \quad \Gamma \vdash \mathrm{P} :\downarrow \forall\mathrm{Y}{:}\mathrm{R}_1.\mathbb{R}_2 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{R}_1 \Rightarrow \mathbb{V}_1 : \_/\delta_1 \quad \Theta;\delta_1\Gamma,\mathrm{X}{:}\mathrm{S}_1,\mathrm{Z}{:}\delta_1\mathbb{R}_2[\mathbb{V}_1/\mathrm{Y}] \vdash \mathrm{Z} \preceq \mathbb{S}_2 \Rightarrow \mathbb{V}_2 : \_/\delta_2}{\Theta;\Gamma \vdash \mathrm{P} \preceq \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \Rightarrow \Lambda\mathrm{X}{:}\mathrm{S}_1.\{1 \rhd \mathrm{Z}{=}\mathrm{P}\langle\delta_2\mathbb{V}_1\rangle, 2{=}\mathbb{V}_2\}.2 : \forall\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2/\delta_2\delta_1|_\Gamma}$$

$$\frac{\mathrm{S}_2 = \{\ldots\} \quad \Gamma \vdash \mathrm{P} :\downarrow \forall\mathrm{Y}{:}\mathrm{R}_1.\mathbb{R}_2 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{R}_1 \Rightarrow \mathbb{V}_1 : \_/\delta_1 \quad \Theta;\delta_1\Gamma,\mathrm{X}{:}\mathrm{S}_1,\mathrm{Z}{:}\delta_1\mathbb{R}_2[\mathbb{V}_1/\mathrm{Y}] \vdash \mathrm{Z} \preceq \mathrm{S}_2 \Rightarrow \mathbb{V}_2 : \_/\delta_2}{\Theta;\Gamma \vdash \mathrm{P} \preceq \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \Rightarrow \lambda(\mathrm{X}{:}\mathrm{S}_1){:}{>}\mathrm{S}_2.(\mathsf{let}\ \mathrm{Z}{=}\mathrm{P}\langle\delta_2\mathbb{V}_1\rangle\ \mathsf{in}\ \mathbb{V}_2 :> \mathrm{S}_2) : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2/\delta_2\delta_1|_\Gamma}$$

$$\frac{\mathrm{S}_2 = \{\ldots\} \quad \Gamma \vdash \mathrm{P} :\downarrow \Pi\mathrm{Y}{:}\mathrm{R}_1.\mathrm{R}_2 \quad \Theta;\Gamma,\mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{X} \preceq \mathrm{R}_1 \Rightarrow \mathbb{V}_1 : \_/\delta_1 \quad \Theta;\delta_1\Gamma,\mathrm{X}{:}\mathrm{S}_1,\mathrm{Z}{:}\delta_1\mathrm{R}_2[\mathbb{V}_1/\mathrm{Y}] \vdash \mathrm{Z} \preceq \mathrm{S}_2 \Rightarrow \mathbb{V}_2 : \_/\delta_2}{\Theta;\Gamma \vdash \mathrm{P} \preceq \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \Rightarrow \lambda(\mathrm{X}{:}\mathrm{S}_1){:}{>}\mathrm{S}_2.(\mathsf{let}\ \mathrm{Z}{=}\mathrm{P}(\delta_2\mathbb{V}_1)\ \mathsf{in}\ \mathbb{V}_2 :> \mathrm{S}_2) : \Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2/\delta_2\delta_1|_\Gamma}$$

Figure 19. Type Inference Rules for Coercive Signature Matching

Constraint Normalization: $\Theta;\Gamma \vdash \Sigma_1 \downarrow (\Sigma_2;\sigma;\delta)$

$$\frac{\begin{array}{c}\forall\mathrm{X}{:}\mathbb{S} \in \Sigma.\ \exists\boldsymbol{\alpha}.\ \Gamma,\Sigma \vdash \mathrm{X}.\mathtt{t} \equiv \boldsymbol{\alpha} : \mathbf{T} \\ \forall\mathrm{X}_1{:}\mathbb{S}_1,\mathrm{X}_2{:}\mathbb{S}_2 \in \Sigma.\ (\mathrm{X}_1 \neq \mathrm{X}_2 \wedge \mathbb{S}_1 \approx \mathbb{S}_2) \Rightarrow \Gamma,\Sigma \vdash \mathrm{X}_1.\mathtt{t} \not\equiv \mathrm{X}_2.\mathtt{t} : \mathbf{T}\end{array}}{\Theta;\Gamma \vdash \Sigma \downarrow (\Sigma;\mathbf{id};\mathbf{id})} \qquad \frac{\Theta;\Gamma \vdash \Sigma_1 \rightsquigarrow (\Sigma_2;\sigma_1;\delta_1) \quad \Theta;\delta_1\Gamma \vdash \Sigma_2 \downarrow (\Sigma_3;\sigma_2;\delta_2)}{\Theta;\Gamma \vdash \Sigma_1 \downarrow (\Sigma_3;\sigma_2\delta_2\sigma_1;\delta_2\delta_1)}$$

Constraint Reduction: $\Theta;\Gamma \vdash \Sigma_1 \rightsquigarrow (\Sigma_2;\sigma;\delta)$

$$\frac{}{\Theta;\Gamma \vdash \Sigma,\mathrm{X}{:}\llbracket\mathfrak{S}(\tau)\rrbracket \rightsquigarrow (\Sigma;\{\mathrm{X} \mapsto [\tau]\};\mathbf{id})} \qquad \frac{\Gamma \vdash \mathbb{S} \downarrow \{\ell_1{:}\mathbb{S}_1,\ldots,\ell_n{:}\mathbb{S}_n\} \quad \mathtt{t} \notin \ell_1,\ldots,\ell_n}{\Theta;\Gamma \vdash \Sigma,\mathrm{X}{:}\mathbb{S} \rightsquigarrow (\Sigma,\mathrm{X}_1{:}\mathbb{S}_1,\ldots,\mathrm{X}_n{:}\mathbb{S}_n;\{\mathrm{X} \mapsto \{\ell_1{=}\mathrm{X}_1,\ldots,\ell_n{=}\mathrm{X}_n\}\};\mathbf{id})}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbb{S} \downarrow \{\mathtt{t}{:}\llbracket\mathfrak{S}(\tau)\rrbracket,\ldots\} \quad \tau \text{ not an } \boldsymbol{\alpha} \\ \mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} :\downarrow \mathbb{S}' \quad \Gamma \vdash \mathbb{S} \equiv \mathbb{S}' \Rightarrow \delta\end{array}}{\Theta;\Gamma \vdash \Sigma,\mathrm{X}{:}\mathbb{S} \rightsquigarrow (\delta\Sigma;\{\mathrm{X} \mapsto \mathrm{P}\};\delta)} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathbb{S} \downarrow \{\mathtt{t}{:}\llbracket\mathfrak{S}(\tau)\rrbracket,\ldots\} \quad \tau \text{ not an } \boldsymbol{\alpha} \\ \mathrm{P} \in \Theta \quad \Gamma \vdash \mathrm{P} :\downarrow \forall\mathrm{Y}{:}\mathrm{S}_1.\mathbb{S}_2 \quad \mathrm{S}_1 \rightrightarrows \exists\overline{\boldsymbol{\alpha}}.\mathbb{S}_1 \quad \Gamma,\mathrm{Y}{:}\mathbb{S}_1 \vdash \mathbb{S} \equiv \mathbb{S}_2 \Rightarrow \delta\end{array}}{\Theta;\Gamma \vdash \Sigma,\mathrm{X}{:}\mathbb{S} \rightsquigarrow (\delta\Sigma,\mathrm{Y}{:}\delta\mathbb{S}_1;\{\mathrm{X} \mapsto \mathrm{P}\langle\mathrm{Y}\rangle\};\delta)}$$

$$\frac{\exists\boldsymbol{\alpha}.\ \Gamma,\Sigma \vdash \mathrm{X}_1.\mathtt{t} \equiv \boldsymbol{\alpha} \equiv \mathrm{X}_2.\mathtt{t} : \mathbf{T} \quad \Gamma \vdash \mathbb{S}_1 \equiv \mathbb{S}_2 \Rightarrow \delta}{\Theta;\Gamma \vdash \Sigma,\mathrm{X}_1{:}\mathbb{S}_1,\mathrm{X}_2{:}\mathbb{S}_2 \rightsquigarrow (\delta\Sigma,\mathrm{X}_1{:}\delta\mathbb{S}_1;\{\mathrm{X}_2 \mapsto \mathrm{X}_1\};\delta)}$$

Figure 20. Constraint Normalization

Top-Level Modules: $\Theta;\Gamma \vdash \mathit{mod} \Rightarrow \mathrm{M} : \mathrm{S}$

Analogous to corresponding module inference rules, but without any mentions of Σ or δ. The only exceptions to this rule are the ones for $\mathsf{canon}(\mathit{sig})$ and $\mathit{mod} :> \mathit{sig}$, which invoke their module-level counterparts directly in the premises:

$$\frac{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}/(\emptyset;\mathbf{id})}{\Theta;\Gamma \vdash \mathsf{canon}(\mathit{sig}) \Rightarrow \mathrm{M} : \mathrm{S}} \qquad \frac{\Theta;\Gamma \vdash \mathit{mod} :> \mathit{sig} \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}/(\emptyset;\mathbf{id})}{\Theta;\Gamma \vdash \mathit{mod} :> \mathit{sig} \Rightarrow \mathrm{M} : \mathrm{S}}$$

Calls to coercive signature matching (in various rules) require empty constraint and $\mathbf{id}$ substitution in the result.

The only new rule, which is identical to the corresponding declarative elaboration rule:

$$\frac{\Theta;\Gamma \vdash \mathrm{P}\ \mathsf{usable} \quad \Theta,\mathrm{P};\Gamma \vdash \mathit{top} \Rightarrow \mathrm{M} : \mathrm{S}}{\Theta;\Gamma \vdash \mathsf{using}\ \mathrm{P}\ \mathsf{in}\ \mathit{top} \Rightarrow \mathrm{M} : \mathrm{S}}$$

Figure 21. Type Inference Rules for Top-Level Modules

We say that an IL kind K is *legal* if it has the form of an EL kind.

We say that an IL signature S is *legal* if:

1. For all signatures within S of the form $[\![\mathrm{K}]\!]$, K is legal.
2. For all signatures within S of the form $\Pi\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$, S_2 must have the form $\{\ldots\}$.
3. For all signatures within S of the form $\forall\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$, either S_2 has the form $\{\ldots\}$ or S_2 has the form $[\![\tau]\!]$. In the latter case, it must be additionally true that S_1 is a valid class (the judgment $\Gamma \vdash \mathrm{S}_1\ \mathsf{class}$ must hold for an appropriate Γ).

Essentially, this definition is just trying to ensure that if we (informally) "ran S through the elaboration judgment for signatures", then it would be accepted. This is not the case if S is an arbitrary IL signature.
We say that a context Γ is *legal* if all constituent signatures and kinds are legal.

We implicitly assume and maintain the invariant that the elaborator and the inference algorithm only deal with legal IL objects, and so we will not mention legality explicitly from here on.

We say that something is *ground* if it has no (free) unification variables.

We say that a signature S is *synthesis* if all signatures within it are ground, with the following exception: signatures within S of the form $[\![\tau]\!]$ or $\forall\mathrm{X}{:}\mathrm{R}.[\![\tau]\!]$ may be non-ground so long as they do not appear in the argument of a functor signature.

We say that a context Γ is *synthesis* if $\forall\alpha{:}\mathrm{K} \in \Gamma.\ \mathrm{K}$ is ground and $\forall\mathrm{X}{:}\mathrm{S} \in \Gamma.\ \mathrm{S}$ is synthesis.

We say that a context $(\Theta;\Gamma)$ is *valid for inference* if $\vdash (\Theta;\Gamma)\ \mathsf{ok}$, Γ is synthesis, and for all $\mathrm{P} \in \Theta$, the signature of P in Γ is ground.

We write $\Gamma' \vdash \delta : \Gamma$ to mean that $\vdash \Gamma'\ \mathsf{ok}$, $\Gamma' \supseteq \delta\Gamma$ and $\forall\boldsymbol{\alpha} \in \mathrm{dom}(\delta).\ \Gamma' \vdash \delta\boldsymbol{\alpha} : \mathbf{T}$.

We write $\Theta;\Gamma \vdash_{\overline{\mathrm{can}}} \sigma : \Sigma$ to mean that $\forall\mathrm{X}{:}\mathbb{S} \in \Sigma.\ \Theta;\Gamma \vdash_{\overline{\mathrm{can}}} \sigma\mathrm{X} : \mathbb{S}$.

Figure 22. Definitions for Soundness

Theorem (Soundness)
Suppose $(\Theta;\Gamma)$ is valid for inference. For all inference judgments that take $\Theta;\Gamma$ as input and return a substitution δ, we have $\delta\Gamma \vdash \delta : \Gamma$.
Suppose further that $\Theta' \supseteq \Theta$, $\Gamma' \vdash \delta' : \delta\Gamma$, $\vdash (\Theta';\Gamma')\ \mathsf{ok}$, and $\Theta';\Gamma' \vdash_{\overline{\mathrm{can}}} \sigma' : \delta'\Sigma$. Then:

1. If $\Gamma \vdash \mathit{meta}_1 \equiv \mathit{meta}_2 \Rightarrow \delta$, then $\Gamma' \vdash \delta'\delta\mathit{meta}_1 \equiv \delta'\delta\mathit{meta}_2$.
2. If $\Theta;\Gamma \vdash \mathit{knd} \Rightarrow \mathrm{K}$, then $\Theta';\Gamma' \vdash \mathit{knd} \leadsto \mathrm{K}$ and K is ground.
3. If $\Theta;\Gamma \vdash \mathit{con} \Rightarrow \mathrm{C} : \mathrm{K}$, then $\Theta';\Gamma' \vdash \mathit{con} \leadsto \mathrm{C} : \mathrm{K}$ and C and K are ground.
4. If $\Theta;\Gamma \vdash \mathit{sig} \Rightarrow \mathrm{S}$, then $\Theta';\Gamma' \vdash \mathit{sig} \leadsto \mathrm{S}$ and S is ground.
5. If $\Theta;\Gamma \vdash \mathit{exp} \Rightarrow e : \tau/(\Sigma;\delta)$ or $\Theta;\Gamma \vdash \mathit{exp} \Rightarrow\!\downarrow e : \tau/(\Sigma;\delta)$, then $\Theta';\Gamma' \vdash \mathit{exp} \leadsto \sigma'\delta' e : \delta'\tau$.
6. If $\Theta;\Gamma \vdash \mathit{exp} : \tau \Rightarrow\!\downarrow e/(\Sigma;\delta)$, $\Gamma \vdash \tau : \mathbf{T}$, and τ is ground, then $\Theta';\Gamma' \vdash \mathit{exp} \leadsto \sigma'\delta' e : \tau$.
7. If $\Theta;\Gamma \vdash \mathit{mod} \Rightarrow \mathrm{M} : \mathrm{S}/(\Sigma;\delta)$ or $\Theta;\Gamma \vdash \mathit{mod} \Rightarrow\!\downarrow \mathrm{M} : \mathrm{S}/(\Sigma;\delta)$, then $\Theta';\Gamma' \vdash \mathit{mod} \leadsto \sigma'\delta'\mathrm{M} : \delta'\mathrm{S}$.
8. If $\Theta;\Gamma \vdash \mathrm{P} \preceq \mathrm{S} \Rightarrow \mathbb{V} : \mathbb{S}/\delta$, $\Gamma \vdash \mathrm{P} : \mathbb{S}'$, $\Gamma \vdash \mathrm{S}\ \ \mathrm{sig}$, and S is ground, then $\Theta';\Gamma' \vdash \mathrm{P} \preceq \mathrm{S} \leadsto \delta'\mathbb{V} : \mathbb{S}$ and $\mathbb{S}$ is ground.
9. If $\Theta;\Gamma \vdash \Sigma_0 \downarrow (\Sigma;\sigma;\delta)$ or $\Theta;\Gamma \vdash \Sigma_0 \rightsquigarrow (\Sigma;\sigma;\delta)$, then $\Theta';\Gamma' \vdash_{\overline{\mathrm{can}}} \sigma'\delta'\sigma : \delta'\delta\Sigma_0$.

Finally: Suppose that $\vdash (\Theta;\Gamma)\ \mathsf{ok}$, Γ is ground, and $\Theta;\Gamma \vdash \mathit{top} \Rightarrow \mathrm{M} : \mathrm{S}$. Then, $\Theta;\Gamma \vdash \mathit{top} \leadsto \mathrm{M} : \mathrm{S}$ and S is ground.

Figure 23. Soundness of the Inference Algorithm w.r.t. Declarative Elaboration