Modular Type Classes

Derek Dryer  
_Toyota Technological Institute at Chicago_

Robert Harper  
_Carnegie Mellon University_

Manuel M.T. Chakravarty  
_University of New South Wales_

Gabriele Keller  
_University of New South Wales_

Follow this and additional works at: http://repository.cmu.edu/compsci

Published In
Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '07, 63-70.
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. Chakravy
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 by 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 is independent of a particular language.

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:

```ml
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:

```ml
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, EqI, associated with its outermost type constructor, \( t \). For example, here is an instance of equality for product types \( t_1 \times t_2 \):

```ml
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:

```ml
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

```ml
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.\footnote{This feature is particularly useful in conjunction with our support for associated types; see Section 2.5.} 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 \textit{A} with additional operations to form a class \textit{B}, at which point \textit{A} is called a \textit{superclass} of \textit{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 \textit{lt} operation. We can do this by first defining an atomic class LT that only supports \textit{lt}, and then defining ORD as a composite of EQ and LT:

\begin{verbatim}
signature ORD = sig
structure E : EQ
structure L : LT
sharing type E.t = L.t
end
\end{verbatim}

The sharing specification makes explicit that ORD is providing two different interpretations of the \textit{same} type, as an equality type and as an ordered type. ORD is an example of what we call a composite \textit{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:

\begin{verbatim}
signature LT = sig
structure E = EqInt, EqProd
structure X : ORD
end
\end{verbatim}

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
(Y.E.eq(x1,x2) andalso Y.L.lt(y1,y2))
end
\end{verbatim}

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

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

\begin{verbatim}
signature ORD = sig
structure E : EqProd(EqInt, EqInt)
structure L = LtProd(structure E = EqInt
structure L = LtInt
end, LtInt)
end
\end{verbatim}

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 \textit{type E.t = int} directly, one would implicitly provide an instance for EQ where \textit{type t = int} through its \textit{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 \textit{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 \textit{int} in Haskell is only permitted to provide a definition for the new operations (namely, \textit{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 \textit{LT}.

2.4 Constrained polymorphism via functors

Under the Harper-Stone interpretation of Standard ML (hereafter, HS) \cite{9}, polymorphic functions in the external (source) language are elaborated into \textit{functors} in an \textit{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 \textit{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 \textit{ad hoc} polymorphic equality function is represented by the functor

\begin{verbatim}
functor eq (X:EQ) :> [X.t * X.t -> bool] = [X.eq]
\end{verbatim}

where the brackets notation describes a module with a single value component. Polymorphic instantiation at a type \textit{\tau} consists of computing a canonical instance of \textit{EQ} where \textit{type t = \tau}, as described above, applying the functor \textit{eq} to \textit{t}, 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 \textit{qualified type} \cite{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 \textit{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

\begin{verbatim}
val eq = overload eq from EQ
\end{verbatim}

Note that there is no need to bind the polymorphic function returned by the overload mechanism to the name \textit{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 \textit{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:

\begin{verbatim}
using EqInt, EqProd in ...
end
\end{verbatim}

\begin{verbatim}
~ \ldots Val(eq(EqProd(EqInt,EqInt))) ((2,3),(4,5))... 
\end{verbatim}

\begin{verbatim}
fun refl y = eq(y,y)
\end{verbatim}

\begin{verbatim}
~ functur refl (X : EQ) => [X.t -> bool] = [\fn y => Val(eq(X)) (y,y)]
\end{verbatim}
(Note: the Val operator seen here is the mechanism in our internal module type system by which a value of type \( \tau \) is extracted from a module of signature \( \Box \).

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

\[
\text{functor Refl = explicit (refl : (X : Eq) \to sig val it : X.t \to bool end)}
\]

we convert the polymorphic function \( \text{refl} \) into an explicit functor \( \text{Refl} \). The programmer can then apply it to an arbitrary module argument of signature \( \text{Eq} \) and project out the component of the result. The reason we require a signature annotation on the explicit construct is that the implicitly-typed \( \text{refl} \) may be declaratively ascribed many different signatures. Whenever \( \text{refl} \) 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 \( \text{Refl} \), she needs to know exactly what shape \( \text{Refl} \)’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:

\[
\begin{align*}
\text{signature COLLECTS = sig} & \text{type t} \\
& \text{type elem} \\
& \text{val empty : t} \\
& \text{val insert : elem \times t \to t} \\
& \text{val member : elem \times t \to bool} \\
& \text{val toList : t \to elem list}
\end{align*}
\]

The distinguished type \( t \) represents the collection type and the associated type component \( \text{elem} \) 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:

\[
\begin{align*}
\text{functor CollectsList (X : Eq) = struct} & \text{type t = X.t list} \\
& \text{type elem = X.t} \\
& \text{val empty = \[\]} \\
& \text{fun insert (x, L) = x::L} \\
& \text{fun member (x, []) = raise NotInCollection} \\
& \quad | \text{member (x, y::L) = X.eq (x, y) \? (member (x, L) \orelse \text{member (x, L)})} \\
& \text{fun toList L = L}
\end{align*}
\]

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:

\[
\begin{align*}
\text{val toList = overload toList from COLLECTS} \\
& \text{fun sumColl C = sum (toList C)}
\end{align*}
\]

The \( \text{sumColl} \) function does not care what type of collection \( C \) is, so long as its element type is \( \text{int} \). Correspondingly, the elaborator will assign \( \text{sumColl} \) the polymorphic type \( \text{(} \forall \tau \text{. functor signature) (X : COLLECTS where type elem = int) \to [X.t \to \text{int}]} \)

Note that the constraint on the type \( X.\text{elem} \) 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 \( \text{sumColl} \).

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 \( \text{assoc} \) in the canonical instance of a type class \( \text{SIG} \) at a given type \( \tau \). This is achieved in our setting via the \( \text{canon}(\text{sig}) \) construct, which we introduced above as a way of explicitly computing a canonical instance. In particular, we can write

\[
\text{canon(SIG where type t = \tau).assoc}
\]

which constructs the canonical instance of \( \text{SIG} \) at \( \tau \) and then projects the \( \text{assoc} \) type from it. In the associated type extension to Haskell, one would instead write \( \text{assoc}(\tau) \).

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, \( \text{assoc} \) would be tied to \( \text{SIG} \)). In contrast, in our setting, it is fine for several different class signatures to have an associated type component called \( \text{assoc} \).

### 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 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 \( \text{EqInt1} \) and \( \text{EqInt2} \) are two observably distinct instances of \( \text{Eq} \) where type \( t = \text{int} \). Consider the following code:

\[
\begin{align*}
\text{structure A = using EqInt1 in} & \text{struct ...fun f x = eq(x,x) ... end} \\
& \text{structure B = using EqInt2 in} \\
& \quad \text{struct ...val y = A.f(3) ... end}
\end{align*}
\]

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 \( \text{EQ} \); in this case, the application \( A.f(3) \) demands an instance of \( \text{Eq} \) where type \( t = \text{int} \), which can only be resolved by \( \text{EqInt2} \). On the other hand, type inference is free to assign the type \( \text{int} \to \text{bool} \) to \( A.f \) at the point where \( f \) is defined, in which case the demand for an instance of \( \text{Eq} \) can only be met by \( \text{EqInt1} \). These are both valid typings, but they lead to observably different behavior.

\[2\text{Note that, due to the principle of phase separation in the ML module system [8], the identity of the assoc type here can be determined purely statically, and elaboration does not actually need to construct the dynamic parts of canon(SIG where type t = \tau)}.\]
2 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 \( A \cdot f \) 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 \( \text{sig type } t \text{ end} \) 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 \( \text{sig type } t \text{ end} \) 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

\[ (X : \text{sig type } a; \text{type } b \text{ end}) \rightarrow [X.a \rightarrow X.b] \]

(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 typed (constructor) module \([C]\) and the term module \([e]\), which contain exactly one component each. These modules are given the signatures \([K]\) and \([-]\), respectively, assuming that \( C \) has kind \( K \),
and e has type τ. Compound modules, or structures, are dependent records of the form
\{ \ell_1 : X_1 : S_1, \ldots, \ell_n : X_n : S_n \}.

Here, \ell_1, \ldots, \ell_n are the pairwise distinct labels, or external names, by which the record components are accessed, and X_1, \ldots, 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 : X_1 : S_1, \ldots, \ell_n : X_n : S_n \},

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

Module signatures are transparent 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 \( S(\tau) \) is definitionally equivalent to \( \tau \). In particular, if this is the kind of a variable, then the variable may be thought of as having \( \tau \) 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 \( \lambda \), must have valuable (pure and terminating) bodies; partial functors, written with a \( \lambda \), impose no restrictions. Consequently, the application of a total functor to a valuable argument, written \( V_1(V_2) \), is itself valuable, whereas application of a partial functor, written \( F(M) \), 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 subtyping 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 \( \text{Val}(M) \) extracts the term component from the module, \( M \), of signature \( [\tau] \). One might expect an analogous operation at the type level, but instead we employ a meta-operation \( \text{Fst}(M) \) that computes the type component of the module \( M \) of signature \( [K] \). When \( M \) is a variable, \( X \), the projection \( \text{Fst}(X) \) is defined to be an associated type variable \( X^\tau \) of kind \( K \). Otherwise, \( \text{Fst}(M) \) is
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 $\text{Fst}(M)$ 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 $\mathbf{T}^n \rightarrow \mathbf{T}$ (short for $\Pi n.1: \mathbf{T}$). In addition, types may be projected (through one or more label projections) only from modules that are variables ($X$) or that have the form $\text{canon}(sig)$. 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, $\text{seal}(P)$, is used to annotate a term with a specific type. Term variables ($\tau$) 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 $\tau$), in which the unconstrained and $\tau$ 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 $\tau$), 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 $\lambda x : \tau_1 \rightarrow \tau_2 \rightarrow \mathbf{T}$ is encodable in our language as $\lambda x : \tau_1 \rightarrow \tau_2 \rightarrow \mathbf{T}$.

The EL module constructs $\text{canon}(sig)$ and $\text{overload} \, \ell s \, \text{from} \, sig \, \mapsto \text{implicit}(P)$ and $\text{explicit}(P : S)$ 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 $\text{implicit}(P)$ and $\text{explicit}(P : S)$ convert between implicit and explicit forms of polymorphic values. The explicit form of a polymorphic value of signature $\forall X.S.[\tau]$ is a functor of signature $\forall X.S.\{X : \tau\}$, 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 $\text{mod} : \tau$, where they are annotated with the signature $\tau$.

### 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 EL, IL, and elaboration.

The main elaboration judgments take as input a context comprised of an IL typing context $\Gamma$ and a canonical instance set $\Theta$. 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 $\{K\}$ to constructors of kind $K$ by applying the meta-operation $\text{Fst}$ described in Section 4.1. Rule 2 computes projections from a $\text{canon}(sig)$ 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 $\tau$. 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: \(\Theta;\Gamma \vdash con \sim C : K\)

\[
\begin{align*}
\frac{\Gamma \vdash P : [K]}{\Theta;\Gamma \vdash P \sim \text{Fst}(P) : K} & \quad (1) \\
\frac{\Theta;\Gamma \vdash \text{canon}(\text{sig}) \sim V : S \quad \Gamma \vdash V, \ell s : [K]}{\Theta;\Gamma \vdash \text{canon}(\text{sig}), \ell s \sim \text{Fst}(V), \ell s : K} & \quad (2)
\end{align*}
\]

Signatures: \(\Theta;\Gamma \vdash \text{sig} \sim S\)

\[
\frac{\Theta;\Gamma \vdash \text{sig}_1 \sim S_1 \quad \Theta;\Gamma;X,S_1 \vdash \text{tsig}_2 \sim S_2 \quad S_2 = \{\ldots\}}{\Theta;\Gamma \vdash \forall X.\text{sig}_1, \text{tsig}_2 \sim \forall X.S_1.S_2} \quad (3)
\]

Terms: \(\Theta;\Gamma \vdash \text{exp} \sim e : \tau\)

\[
\frac{x : \tau \in \Gamma}{\Theta;\Gamma \vdash x \sim x : \tau} \quad (5)
\]

\[
\frac{\Gamma \vdash \tau_1 : T \quad \Theta;\Gamma, x.\tau_1 \vdash \text{exp} \sim e : \tau_2}{\Theta;\Gamma \vdash \lambda x.\tau \sim \lambda x.\tau_1 : e \sim \tau_2} \quad (6)
\]

\[
\frac{\Gamma \vdash P : [\tau] \quad \Theta;\Gamma \vdash \text{Val}(P) : \tau}{\Theta;\Gamma \vdash P \sim \text{Val}(P) : \tau} \quad (8)
\]

\[
\frac{\Theta;\Gamma \vdash \text{mod} \sim M : S \quad \Theta;\Gamma;X,S \vdash \text{exp} \sim e : \tau \quad X \notin \text{FV}(\tau)}{\Theta;\Gamma \vdash \text{let}\ X = \text{mod} \ in \ \text{exp} \ \sim \ \text{let}\ X = \text{M} \ in \ e : \tau} \quad (11)
\]

Modules: \(\Theta;\Gamma \vdash \text{mod} \sim M : S\)

\[
\frac{\Theta;\Gamma \vdash \text{exp} \sim e : \tau \quad \Theta;\Gamma \vdash [\text{exp}] \sim [e] : [\tau]}{\Theta;\Gamma \vdash \text{X \notin \text{FV}(exp) \ 
\quad \Gamma \vdash \text{class} X : S \sim \Theta' \quad \Theta',\Theta';\Gamma;X,S \vdash \text{exp} \sim v : \tau}} \quad (14)
\]

\[
\frac{\Theta;\Gamma \vdash \text{sig} \sim S_1 \quad \Theta;\Gamma;X,S_1 \vdash \text{mod} \sim V : S_2 \quad S_2 = \{\ldots\}}{\Theta;\Gamma \vdash \lambda X.\text{sig} \sim \lambda X.S_1.V : \forall X.S_1.S_2} \quad (15)
\]

\[
\frac{\Theta;\Gamma \vdash \text{sig} \sim S \quad \Gamma \vdash S \sim S \quad \Theta;\Gamma \vdash \text{class} V : S}{\Theta;\Gamma \vdash \text{overload} \ \ell s \ \text{from} \ \text{sig} \ \sim \ \lambda X.\text{S}.(X.\ell s : [\tau]) : \forall X.S.\tau} \quad (18)
\]

Coercive Signature Matching: \(\Theta;\Gamma \vdash P \sim S \sim V : S\)

\[
\frac{\Gamma \vdash P : [K] \quad \Gamma \vdash K \sim K}{\Theta;\Gamma \vdash P \sim [K] \sim P : [K]} \quad (20)
\]

\[
\frac{\text{S is of the form } [\tau] \text{ or } \forall X.S.\tau}{\Theta;\Gamma \vdash P \sim S \sim V : S} \quad (21)
\]

Top-Level Modules: \(\Theta;\Gamma \vdash \text{top} \sim M : S\)

\[
\frac{\Theta;\Gamma \vdash P \text{ usable} \quad \Theta;\Gamma \vdash \text{top} \sim M : S}{\Theta;\Gamma \vdash \text{using} \ P \ \text{in} \ \text{top} \ \sim M : S} \quad (22)
\]

Canonical Modules: \(\Theta;\Gamma \vdash \text{class} V : S\)

\[
\frac{\Gamma \vdash \tau : T}{\Theta;\Gamma \vdash [\tau] : [\text{5}(\tau)]} \quad (23)
\]

\[
\frac{\Gamma,\alpha.T^n \vdash \tau : T}{\Theta;\Gamma \vdash [\alpha:T^n,\tau] : [\text{Hrn} : T^n,\text{5}(\tau)]} \quad (24)
\]

\[
\frac{\forall i \in 1..n \quad \Theta;\Gamma \vdash \text{class} V_i : S_i \quad t \notin \{t_1,\ldots,t_n\}}{\Theta;\Gamma \vdash \text{class} \{t_1 = V_1,\ldots,t_n = V_n\} = \{t_1:S_1,\ldots,t_n:S_n\}} \quad (25)
\]

\[
\frac{P \in \Theta \quad \Gamma \vdash P : S \quad S = \{t : [\text{5}(\tau)],\ldots\}}{\Theta;\Gamma \vdash P : S} \quad (26)
\]

\[
\frac{\Theta;\Gamma \vdash \text{class} V : S \quad \Gamma \vdash S_1 \leq S_2 \quad \Gamma \vdash P \sim S_2[V/X]}{\Theta;\Gamma \vdash P[V] : S_2[V/X]} \quad (27)
\]

Class Elaboration: \(\Gamma \vdash S \text{ class} \quad \Theta \vdash \text{class} X : S \sim \Theta\)

\[
\frac{\Gamma \vdash S \text{ sig} \quad S' \ \text{params}(S') \subseteq \{\pi\} \quad \Gamma,\pi,X,S \vdash \text{paths}(X:S) \vdash \Theta}{\Gamma \vdash \text{class} X : S \sim \Theta} \quad (30)
\]

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 \( \lambda \)-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 \( \forall X: S . \tau \), 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 non-deterministic. Correspondingly, what the second premise does is to pick a transparent subsignature \( S' \) of \( S \) that realizes these parameters with some choices \( \tau_1, \ldots, \tau_n \). Then, the third premise finds the canonical module \( V \) of signature \( S' \) using the canonical module judgment \( \Theta; \Gamma \vdash_{\text{can}} V : S' \), 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 \( \forall X: S . \tau \) to assign to \( \exp \). Suppose that \( S \) is an atomic class signature like \( \mathbb{E} \). In order to see whether \( \exp \) can be elaborated with this type, we add the class constraint \( X: S \) to the context and make it a canonical instance of the signature \( S \) where type \( t = X . t \) (by adding \( X \) to \( \Theta \)) before type-checking \( \exp \). The last step is critical: if \( X \) is not added to \( \Theta \), 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 \( \Theta \). To simplify the formalization of other judgments, we require all the instance structures in \( \Theta \) to have atomic signature. Thus, in general we need a way of parsing the class constraint \( X: S \) in order to produce a set of paths \( \Theta' \) (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 \( \Gamma \vdash_{\text{class}} X : S \rightsquigarrow \Theta' \) 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 \( \text{ORD} \) from Section 2.3, then \( \Theta' \) would be the set \( \{ X.E, X.L \} \). Note that the instances in \( \Theta' \) are guaranteed not to overlap with any instances in the input instance set \( \Theta \) because the instances in \( \Theta' \) 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 \( P_1(P_2) \) by matching the argument \( P_2 \) against \( P_1 \)'s argument signature \( S_1 \) via the coercive signature matching judgment, \( \Theta; \Gamma \vdash P_2 \preceq S_1 \rightsquigarrow \Sigma; V : S_1 \). 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 \( S_1 \) describing \( V \) is not used in this particular rule, it is guaranteed to be a subtype of the target signature \( S_1 \).

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

Rules 18 and 19 translate the overload \( \ell \)s from \( \text{sig} \) mechanism essentially as prescribed in Section 2.4. The latter rule is useful for overloading a value component of \( \text{sig} \) that already has a polymorphic type. The class constraint on that value component is joined with \( \text{sig} \) 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 \( [K] \), the input path \( P \) must be an atomic type module whose component is of kind \( K \). To coerce to a (potentially polymorphic) type signature \( S \), it must be the case that \( P \) has a more general polymorphic type than \( S \). This subsumption check is captured very concisely by checking whether the \( \eta \)-expansion of \( P \) (with respect to constrained type abstraction) can be assigned the signature \( S \).

Rule 22 elaborates using \( P \) in \( \text{top} \) by first checking whether \( P \) is \( \text{usable} \) and then adding \( P \) to the canonical instance set \( \Theta \) during the elaboration of \( \text{top} \). Usability is determined by the \( \text{usable instance judgment} \Theta; \Gamma \vdash P \text{ usable} \) (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 \( \Theta \).

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 \( \Theta \)) or the result of applying a canonical instance functor from \( \Theta \) to a canonical argument. Canonical modules may also contain arbitrary unconstrained type components (named something other than \( \tau \)).

Finally, Rule 30 defines the class elaboration judgment, written \( \Gamma \vdash_{\text{class}} X : S \rightsquigarrow \Theta \), which checks whether \( S \) is a class signature and then parses the class constraint \( X: S \) 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 \( \sigma \) corresponding to the abstract type components of \( S \). It also returns \( S \), a transparent version of \( S \) with the abstract type specifications replaced by references to \( \pi \). For example, \( [T] \vdash_{\Sigma} \exists \alpha . [S(\alpha)] \). The third premise normalizes \( S \) to \( S' \) (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 \( \tau \) components together with the unconstrained components) transparently equal to one of the \( \pi \). 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 equation \( \mathbb{E} \) with a shared \( t \) component. Such an \( S \) is a perfectly legitimate class, but the \( \Theta' \) 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 \( \mathcal{V} \) [3]. In particular, we thread through the inference rules a substitution \( \delta \) whose domain consists of unification variables, denoted by bold \( \alpha \).

In addition, polymorphic instantiation in the presence of type classes generates *constraints*, which we denote \( \Sigma \). Constraints are sets of \( X:S \) bindings, in which the \( X \)’s do not appear free in the \( S \)’s. Each \( X:S \) represents a demand generated by the algorithm for
\[
\begin{align*}
\text{partition}(\bar{\alpha};\Sigma) & \triangleq (\Sigma_1;\Sigma_2), \\
\text{where } \Sigma_2 = \{X:S \mid X:S \in \Sigma \land \exists \alpha \in \bar{\alpha}. S = \{t:S(\alpha), \ldots\}\} \text{ and } \Sigma_1 = \Sigma - \Sigma_2 \\
\text{makesig}(\bar{\alpha};\Sigma) & \triangleq \{\text{tyvars} Y: \{1:T, \ldots, m:T\}, \text{consts} \{1:S_1', \ldots, n:S_n'\}\}, \\
\text{where } \bar{\alpha} = \alpha_1, \ldots, \alpha_m \text{ and } \Sigma = X_1:S_1, \ldots, X_n:S_n \text{ and } S_i' = S_i[Y,j/\alpha_j]_{j=1}^n \\
\text{genvars}(\Gamma;\Sigma;\tau) & \triangleq \bar{\alpha}_2, \\
\text{where } \bar{\alpha}_2 \text{ is the greatest set such that } \bar{\alpha}_1 \cup \bar{\alpha}_2 = \text{UV}(\Sigma,\tau) \text{ and } \bar{\alpha}_1 \cap \bar{\alpha}_2 = \emptyset \\
\text{and } \bar{\alpha}_2 \cap \text{UV}(\Gamma,\Sigma_1) = \emptyset, \text{ where partition}(\bar{\alpha}_2;\Sigma) = (\Sigma_1;\Sigma_2)
\end{align*}
\]

**Type Unification:**
\[
\Gamma \vdash \tau_1 \equiv \tau_2 \Rightarrow \delta \\
\Gamma \vdash \tau_1 \Downarrow \tau_2 \Rightarrow \delta \\
\Gamma \vdash \tau_1 \equiv \tau_2 \Rightarrow \delta
\]

**Terms:**
\[
\Theta; \Gamma \vdash \exp \Rightarrow e : \tau/(\Sigma;\delta)
\]

**Terms With Constraint and Type Normalization:**
\[
\Theta; \Gamma \vdash \exp \Rightarrow e : \tau'/\{\Sigma;\delta\} \\
\Theta; \Gamma \vdash \exp \Rightarrow e : \tau/(\Sigma;\delta) \\
\Theta; \Gamma \vdash \exp \Rightarrow \{e\} : \tau /\{(\tau_1;\delta_1)\}
\]

**Modules:**
\[
\Theta; \Gamma \vdash \text{mod} \Rightarrow M : S/(\Sigma;\delta)
\]

**Constraint Normalization:**
\[
\Theta; \Gamma \vdash \Sigma \vdash (\Sigma;\delta) \Rightarrow (\Sigma;\delta) \\
\Theta; \Gamma \vdash \Sigma \vdash (\Sigma;\delta) \Rightarrow (\Sigma;\delta) \\
\Theta; \Gamma \vdash \Sigma \vdash (\Sigma;\delta) \Rightarrow (\Sigma;\delta)
\]

**Constraint Reduction:**
\[
\Theta; \Gamma \vdash \Sigma \vdash (\Sigma;\delta) \\
\Theta; \Gamma \vdash \Sigma \vdash (\Sigma;\delta) \\
\Theta; \Gamma \vdash \Sigma \vdash (\Sigma;\delta)
\]

**Figure 4. Key Type Inference Rules**
a canonical module of signature $\Sigma$ 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 $\Rightarrow$ 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 $\alpha$ with a type $\tau$ that is only valid in a later scope. The premise $\Rightarrow \Gamma[\alpha/\alpha]$ok safeguards against this.

Rules 33–35 are standard, but there are a few points of note. First, the $\alpha$ 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 $\Sigma$'s and threading $\delta$'s. In the conclusion of the rule, the output substitution $\delta$ is subjected to an operation written $\Sigma_e$. This restricts the domain of $\delta$ to UV($\Gamma$) (the free unification variables of $\Gamma$). It is essentially a form of garbage collection that removes from $\delta$ 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 $\Delta$.

The premises of these rules employ a variant of inference written with $\Rightarrow\ldots$. 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 $\tau$ component is equal to a unification variable. Certain rules (such as Rule 39 for generalization) require their premises to have their $\tau$ components 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 $\forall X.\Sigma[\tau]$, it uses the judgment $\Sigma \Rightarrow X.\Sigma[\tau]$ to generate fresh unification variables $\overline{\alpha}$ corresponding to the abstract type components of $\Sigma$. It then applies $P$ to an unknown canonical module $X$ of signature $\delta$, and projects out the value component. This in turn effects a demand for $X.\Sigma$ in the output constraint. For example, if $\Sigma$ were the class $\text{Eq}$, then the output constraint would be $X::\text{Eq}$ where type $\tau = \alpha$. (Note: the “$\Rightarrow\ldots$” judgment used in the first premise indicates $\forall X.\Sigma[\tau]$ is the normal form signature of $P$, and the last premise normalizes $\tau$ so that references to type components of $X$ become references to the corresponding $\overline{\alpha}$.)

Rule 39 performs polymorphic generalization. The first premise translates $exp$ to a valuable term $v$ with type $\tau$, and generates a normalized constraint $\Sigma$. The second premise calculates the largest set of variables $\overline{\alpha}$ over which $v$ may be abstracted. Based on $\overline{\alpha}$, the third premise partitions $\Sigma$ accordingly into $\Sigma_2$ (which will join $\overline{\alpha}$ in the abstraction) and $\Sigma_1$ (which will propagate out of the rule). Essentially, $\Sigma_2$ comprises the constraints that refer to variables in $\overline{\alpha}$ and $\Sigma_1$ comprises the constraints that do not. Finally, we use the makesig macro to combine $\overline{\alpha}$ and $\Sigma_2$ into a class signature $\delta$ that we can abstract $v$ over. The remainder of the premises are simply doing namespace management to convert references to $\overline{\alpha}$ and $\text{dom}(\Sigma_2)$ into projections from the module variable $X$.

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

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:

If $\Theta; \Gamma \vdash \Sigma_1 \dfeq (\Sigma_2; \sigma; \delta)$, then $\forall X.\Sigma \in \Sigma_1, \Theta; \text{dom}(\Sigma_2); \delta; \Sigma_2 \vdash \sigma X : \delta S$.

That is, if we treat the domain of the normalized constraint $\Sigma_2$ as a set of canonical instances, then from those instances together with the canonical instances already in $\Theta$, the substitution $\sigma$ shows how to construct canonical modules to satisfy all the demands of the original constraint $\Sigma_1$ (subject to type substitution $\delta$).

To make this concrete, suppose $\Theta$ contains the $\text{EqInt}$ and $\text{EqProd}$ instance modules given in Section 2.1, and suppose that $\Sigma_1$ is $X::\text{Eq}$ where type $\tau = \text{int} + \alpha$. Then the normalized $\Sigma_2$ would be $Y::\text{Eq}$ where type $\tau = \alpha$, and the substitution $\sigma$ would map $X$ to $\text{EqProd}(\text{EqInt}, Y)$. (In this case, $\delta$ would simply be the identity substitution $\text{id}$.)

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 $\tau$ 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, $\Theta' \subseteq \Theta$, $\Gamma' \dfeq \delta' : \delta \Gamma$, $\Theta; \Gamma' \vdash \delta' : \delta \Gamma$, and $\forall X.\Sigma \in \Sigma, \Theta'; \Gamma' \vdash \sigma X : \delta' S$. Then:

1. If $\Theta; \Gamma \vdash \exp \dfeq \epsilon : \tau/\Sigma, \delta$, then $\Theta'; \Gamma' \vdash \exp \sim \sigma' \delta' : \delta' \tau$.

2. If $\Theta; \Gamma \vdash \text{mod} \dfeq \text{M} : \text{S} : \Sigma, \delta; \delta'$, then $\Theta'; \Gamma' \vdash \text{mod} \sim \sigma' M : \delta' S$.

Consider part 1. Informally, $\Theta$, $\Gamma$, and $\exp$ are inputs. If type inference on $exp$ succeeds, it produces an IL term $\epsilon$, along with a constraint $\Sigma$ and a substitution $\delta$. If in any “future world” ($\Theta'; \Gamma'$) the constraint $\Sigma$ can be solved by substitutions $\sigma'$ and $\delta'$, then $\exp$ will declaratively elaborate to $\sigma' \delta' \epsilon$ 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:

Note: The page contains a complex text discussing type inference and constraint normalization in a system similar to Haskell. The text includes formal proofs and examples, typical of a research paper on type systems.
Given this overloading, the expression `show (read("1"))` is well-typed because `f` may be assigned the type `X.t -> 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 `F`, 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 functions, 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^4 that integrates concepts, which are closely related to type classes, into System F. However, F^4 does not support type inference. Siek’s thesis [21] defines a related language G, which supports inference for type applications, but not type abstractions. Concepts in G are treated as a distinct construct, unrelated to modules, and 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


The judgments $\Gamma \vdash K_1 \leq K_2$ and $\Gamma \vdash S_1 \equiv S_2$ 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)
Terms: $\Theta; \Gamma \vdash exp \sim e : \tau$

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \tau \in \Gamma$</td>
<td>$\Theta; \Gamma \vdash x \sim x : \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau_1 : \Theta$</td>
<td>$\Lambda; \Gamma, x : \tau_1 \vdash exp \sim e : \tau_2$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \lambda x. exp \sim \lambda x. \tau_1 : e : \tau_1 \rightarrow \tau_2$</td>
<td>$\Theta; \Gamma \vdash exp_1 \sim e_1 : \tau_2 \rightarrow \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \lambda x. exp \sim \lambda x. \tau_1 : e : \tau_1 \rightarrow \tau_2$</td>
<td>$\Theta; \Gamma \vdash exp_2 \sim e_2 : \tau_2$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash P : \tau$</td>
<td>$\Theta; \Gamma \vdash P : \lambda \tau. \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \lambda \tau. \tau$</td>
<td>$\Theta; \Gamma \vdash P \sim Val(P) : \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash Val(P) : \tau$</td>
<td>$\Theta; \Gamma \vdash P \sim Val(P) : \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash mod \sim M : S$</td>
<td>$\Theta; \Gamma \vdash Mod \sim M : S$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash X:S \vdash exp \sim e : \tau$</td>
<td>$\Theta; \Gamma \vdash Mod \sim M : S$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash tyyp \sim \tau : T$</td>
<td>$\Theta; \Gamma \vdash exp \sim e : \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash exp \sim tyyp \sim \tau : T$</td>
<td>$\Theta; \Gamma \vdash exp \sim e'$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash exp \sim e$</td>
<td>$\Theta; \Gamma \vdash exp \sim e : \tau$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash exp \sim e$</td>
<td>$\Theta; \Gamma \vdash exp \sim e : \tau$</td>
</tr>
</tbody>
</table>

Figure 7. Elaboration Rules for Terms

Kinds: $\Theta; \Gamma \vdash knrd \sim K$

<table>
<thead>
<tr>
<th>Type Constructor</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta; \Gamma \vdash knrd \sim knrd$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
<tr>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
<td>$\Theta; \Gamma \vdash \tau : T \rightarrow T$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Γ ⊢ P : S$</td>
<td>$Γ ⊢ P \rightsquigarrow P : S$</td>
</tr>
<tr>
<td>$Γ \vdash \text{con} \rightsquigarrow C : K$</td>
<td>$Γ \vdash \text{con} \rightsquigarrow [C] : [K]$</td>
</tr>
<tr>
<td>$Γ \vdash \text{exp} \rightsquigarrow e : τ$</td>
<td>$Γ \vdash \text{exp} \rightsquigarrow [e] : [τ]$</td>
</tr>
<tr>
<td>$X ∉ \text{FV(exp)}$</td>
<td>$Γ \vdash \text{class} X : S \rightsquigarrow Θ'$</td>
</tr>
<tr>
<td>$Θ, Θ'; Γ, X. S \vdash \text{exp} \rightsquigarrow v : τ$</td>
<td>$Γ \vdash \text{mod} \rightsquigarrow M : S$</td>
</tr>
<tr>
<td>$Γ \vdash \text{mod} \rightsquigarrow M : S$</td>
<td>$Γ \vdash \text{mod} \rightsquigarrow M : S$</td>
</tr>
<tr>
<td>$Γ \vdash \text{sig} \rightsquigarrow S_1$</td>
<td>$Γ \vdash \text{sig} \rightsquigarrow S_1$</td>
</tr>
<tr>
<td>$Θ, Γ, X. S_1 \vdash \text{mod} \rightsquigarrow M : S_2$</td>
<td>$Θ, Γ, X. S_1 \vdash \text{mod} \rightsquigarrow M : S_2$</td>
</tr>
<tr>
<td>$S_2 = {\ldots}$</td>
<td>$S_2 = {\ldots}$</td>
</tr>
<tr>
<td>$Γ \vdash \text{mod} \rightsquigarrow M : S$</td>
<td>$Γ \vdash \text{mod} \rightsquigarrow M : S$</td>
</tr>
<tr>
<td>$Γ \vdash \text{sig} \rightsquigarrow S$</td>
<td>$Γ \vdash \text{sig} \rightsquigarrow S$</td>
</tr>
<tr>
<td>$Γ \vdash \text{con} \rightsquigarrow C : K$</td>
<td>$Γ \vdash \text{con} \rightsquigarrow C : K$</td>
</tr>
<tr>
<td>$Γ \vdash \text{exp} \rightsquigarrow e : τ$</td>
<td>$Γ \vdash \text{exp} \rightsquigarrow e : τ$</td>
</tr>
<tr>
<td>$X ∉ \text{FV(exp)}$</td>
<td>$X ∉ \text{FV(exp)}$</td>
</tr>
</tbody>
</table>

**Figure 8.** Elaboration Rules for Modules
Coercive Signature Matching: \( \Theta; \Gamma \vdash P \leq S \sim V : S \)

\[
\begin{align*}
\frac{\Gamma \vdash P : [K]}{\Gamma \vdash [K] \sim P : [K]} \\
\frac{\Theta; \Gamma \vdash P \leq [K] \sim P : [K]}{\Theta; \Gamma \vdash P \leq S \sim V : S}
\end{align*}
\]

\( S \) is of the form \([\tau]\) or \(\forall X:S.\tau\)

\( \Theta; \Gamma \vdash [P] \sim V : S \)

\( \Gamma \vdash P : \{\ldots\} \)

\( \Theta; \Gamma \vdash P, \ell_1 : S_1 \vdash S_1 \sim S_1 \sim V_1 : S_1 \)

\( \Gamma, X_1; S_1 \vdash P \leq \{\ell_1 \triangleright X_1, \ell_1 \triangleright X : S\} \sim \{\ell_1 \triangleright X_1 = V_1, \ell_1 \triangleright X = V\} : \{\ell_1 \triangleright X_1, \ell_1 \triangleright X : S\} \)

\( S_2 = \{\ldots\} \)

\( \Gamma \vdash P : \forall X:S_1.\forall S_2 \sim \lambda X:S_1 : S_2 \vdash \lambda (X:S_1) : S_2 \vdash Z \leq S_2 \sim V_2 : \_
\)

\( \Theta; \Gamma \vdash P \leq \forall X:S_1, S_2 \sim \lambda X:S_1 : S_2 \vdash \lambda (X:S_1) : S_2 \vdash Z \leq S_2 \sim V_2 : \_
\)

\( \Theta; \Gamma \vdash P \vdash \forall X:S_1, S_2 \sim \lambda (X:S_1) : S_2 \vdash (let \ Z=P(V_1) \ in \ V_2 : S_2) : \Pi X:S_1, S_2 \)

Figure 9. Elaboration Rules for Coercive Signature Matching

Top-Level Modules: \( \Theta; \Gamma \vdash top \sim M : S \)

Analogous to module elaboration rules, minus the rule for \([\text{exp}]\), but plus:

\[
\Theta; \Gamma \vdash P \text{ usable} \quad \Theta; P; \Gamma \vdash top \sim M : S \\
\Theta; \Gamma \vdash P \text{ using } P \text{ in } top \sim M : S
\]

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

\[
\vdash (\Theta; \Gamma) \text{ ok} \\
\vdash (\Theta; P; \Gamma) \text{ ok}
\]

\( \vdash (\Theta; P; \Gamma) \text{ ok} \)

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

Canonical Modules: \( \Theta; \Gamma_{\text{can}} V : S \)

\[
\begin{align*}
\frac{\Gamma \vdash \tau : T}{\Theta; \Gamma_{\text{can}} [\tau] : [S(\tau)]} \\
\frac{\Theta; \Gamma_{\text{can}} [\lambda \alpha:T^n.\tau] : [S(\tau)]}{\Theta; \Gamma_{\text{can}} (\lambda \alpha:T^n.\tau) : [\Pi \alpha:T^n.\tau] : [S(\tau)]} \\
\frac{\forall i \in 1..n : \Theta; \Gamma_{\text{can}} V_i : S_i \quad i \notin \{1, \ldots, n\}}{\Theta; \Gamma_{\text{can}} (\ell_1 = V_1, \ldots, \ell_n = V_n) : [S(\tau)]} \\
\frac{P \in \Theta \quad \Gamma \vdash P : S \quad S = \{t,S(\tau)\}_{t \in \Theta} \quad P \in \Theta \\
\quad \Theta; \Gamma_{\text{can}} P : S}{\Theta; \Gamma_{\text{can}} P : S} \\
\frac{P \in \Theta \quad \Gamma \vdash P : \forall X:S_1, S_2 \quad \Theta; \Gamma_{\text{can}} V : S_1 \quad \Gamma \vdash S_1 \leq S_1}{\Theta; \Gamma_{\text{can}} P(V) : S_2[V/X]} \\
\frac{\Theta; \Gamma_{\text{can}} V : S \quad \Gamma \vdash S \equiv S}{\Theta; \Gamma_{\text{can}} V : S}
\end{align*}
\]

Figure 11. Elaboration Rules for Canonical Modules
Signature Similarity: \( R \approx S \)

\[
R = \{ \ell_1:x_1:R_1, \ldots, \ell_n:x_n:R_n \} \quad S = \{ \ell_1:y_1:S_1, \ldots, \ell_n:y_n:S_n \}
\]

Abstract Type Extraction: \( S \Rightarrow \exists ! S \)

\[
S \Rightarrow \emptyset. \quad [T] \Rightarrow \exists \alpha. [S(\alpha)]
\]

Class Parameters: \( \text{params}(S) \)

\[
\text{params}([S(\alpha)]) \overset{\text{def}}{=} \{ \alpha \}
\]

\[
\text{params}(\{\}) \overset{\text{def}}{=} \emptyset
\]

\[
\text{params}(\{ \ell_1:x_1:S_1, \ell_2:x_2:S_2 \}) \overset{\text{def}}{=} \text{params}(S_1) \cup \text{params}(\{ \ell_2:x_2:S_2 \}) \quad \text{where } t \notin \{ \ell_1, \ell_2 \}
\]

Paths to Atomic Instance Modules in a Class: \( \text{paths}(P : S) \)

\[
\text{paths}(P : [S(\tau)]) \overset{\text{def}}{=} \emptyset
\]

\[
\text{paths}(P : \{\}) \overset{\text{def}}{=} \emptyset
\]

\[
\text{paths}(P : \{ \ell_1:x_1:S_1, \ell_2:x_2:S_2 \}) \overset{\text{def}}{=} \text{paths}(P, \ell_1 : S_1) \cup \text{paths}(P : \{ \ell_2:x_2:S_2 \}) \quad \text{where } t \notin \{ \ell_1, \ell_2 \}
\]

\[
\text{paths}(P : \{ t:x:[S(\tau)], \ldots \}) \overset{\text{def}}{=} \{ P \}
\]

Instance Set Simplification: \( \Gamma \vdash \Theta \downarrow \Theta' \)

\[
\forall P_1 \in \Theta, \exists P_2 \in \Theta'. \Gamma \vdash P_1 : S_1 \land \Gamma \vdash P_2 : S_2 \land \Gamma \vdash S_1 \equiv S_2
\]

\[
\forall P_1, P_2 \in \Theta'. (P_1 \neq P_2 \land \Gamma \vdash P_1 : S_1 \land \Gamma \vdash P_2 : S_2 \land S_1 \equiv S_2) \Rightarrow \Gamma \vdash P_1, t \neq P_2, t : T
\]

Class Elaboration: \( \Gamma \vdash _\text{class} X : S \rightarrow \Theta \)

\[
\frac{\Gamma \vdash S \quad \text{sig} \quad S \Rightarrow \exists ! S \quad \Gamma, \alpha \vdash S \downarrow S' \quad \text{params}(S') \subseteq \{ \alpha \} \quad \Gamma, \alpha, X : S \vdash \text{paths}(X : S) \downarrow \Theta}{\Gamma \vdash _\text{class} X : S \rightarrow \Theta}
\]

Well-formed Classes: \( \Gamma \vdash S \text{ class} \)

\[
\Gamma \vdash _\text{class} X : S \rightarrow \Theta
\]

\[
\frac{\Gamma \vdash S \text{ class}}{\Gamma \vdash S \text{ class}}
\]

Figure 12. Elaboration Rules for Classes
**Concrete Signatures:** \( \Gamma \vdash S \) concrete

<table>
<thead>
<tr>
<th>( \Gamma \vdash \tau : T )</th>
<th>( \forall S \in S, \Gamma \vdash S ) concrete ( t \not\in \tau )</th>
<th>( \Gamma \vdash \tau : T )</th>
<th>( \Gamma \vdash { t \circ X : [\tau] } ) concrete</th>
<th>( \Gamma \vdash S ) concrete</th>
<th>( \Gamma \vdash S ) concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash S ) concrete</td>
<td>( \Gamma \vdash S ) concrete</td>
<td>( \Gamma \vdash S ) concrete</td>
<td>( \Gamma \vdash S ) concrete</td>
<td>( \Gamma \vdash S ) concrete</td>
<td></td>
</tr>
</tbody>
</table>

**Instances:** \( \Gamma \vdash (P; p) \) instance

\[
\begin{align*}
\Gamma \vdash P : S & \quad S = \{ t \circ Y : [\delta(C)] \} \ldots \quad \Gamma \vdash C \upharpoonright p \\
\Gamma \vdash (P; p) \) instance \\
\Gamma \vdash P : \forall X:S_1,S_2 \quad \Gamma \vdash S_1 \text{ class} & \quad S_1 = \exists \pi.S_1, S_2 = \{ t \circ Y : [\delta(C)] \} \ldots \\
\Gamma, \pi \vdash S_1 \upharpoonright S_1' & \quad \text{params}\{S_1'\} = \{ \beta \} \quad \Gamma, \pi, X:S_1 \vdash C \upharpoonright p(\beta) \\
\Gamma \vdash (P; p) \) instance
\end{align*}
\]

**Usable Instances:** \( \Theta; \Gamma \vdash P \) usable

\[
\begin{align*}
\Gamma \vdash (P; p) \) instance & \quad \Gamma \vdash P : S \quad \forall P' \in \Theta, (\Gamma \vdash (P'; p) \) instance \wedge \Gamma \vdash P' : S' \Rightarrow S \not\approx S' \\
\Theta; \Gamma \vdash P \) usable \\
\Gamma \vdash (P; p) \) instance & \quad \Gamma \vdash P : \forall X:S_1,S_2 \quad \forall P' \in \Theta, (\Gamma \vdash (P'; p) \) instance \wedge \Gamma \vdash P' : \forall X:S_1'.S' \Rightarrow S \not\approx S' \\
\Theta; \Gamma \vdash P \) usable
\end{align*}
\]

**Figure 13.** Elaboration Rules for Instances
Analogous to corresponding declarative elaboration rules, except:
\[
\Theta; \Gamma \vdash \text{canon}(\text{sig}) \Rightarrow \bigvee \forall \ell \cdot \ell / (\emptyset; \text{id}) \quad \Theta; \Gamma \vdash \forall \ell.x : [\ell ; [K]
\]

Figure 15. Type Inference Rules for Kinds, Constructors and Signatures
Terms With Constraint and Type Normalization: $\Theta; \Gamma \vdash exp \Rightarrow e : \tau/(\Sigma; \delta)$

\[
\begin{align*}
\Theta; \Gamma \vdash exp \Rightarrow e : \tau'/(\Sigma_2; \delta_1) & \quad \Theta; \delta_1 \Gamma \vdash \Sigma_1 \downarrow (\Sigma_2; \sigma; \delta_2) \quad \delta_2 \delta_1 \Gamma \vdash \delta_2 \tau' \downarrow \tau \\
\Theta; \Gamma \vdash exp \Rightarrow \sigma \delta_2 e : \tau/(\Sigma_2; \delta_2 \delta_1[v])
\end{align*}
\]

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

\[
\begin{align*}
\Theta; \Gamma \vdash exp \Rightarrow e : \tau'/(\Sigma_1; \delta_1) & \quad \delta_1 \Gamma \vdash \tau' \equiv \tau \Rightarrow \delta_2 \quad \Theta; \delta_2 \delta_1 \Gamma \vdash \delta_2 \Sigma_1 \downarrow (\Sigma_2; \sigma; \delta_3) \\
\Theta; \Gamma \vdash exp : \tau \Rightarrow \sigma \delta_2 \delta_1 e/(\Sigma_2; \delta_2 \delta_1[v])
\end{align*}
\]

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

\[
\begin{align*}
\text{partition}(\overline{\alpha}; \Sigma) & \triangleq (\Sigma_1; \Sigma_2), \quad \text{where } \Sigma_2 = \{X:S \mid X:S \in \Sigma \land \exists \alpha \in \overline{\alpha}. S = \{t:S(\alpha) \ldots \} \} \text{ and } \Sigma_1 = \Sigma - \Sigma_2 \\
\text{makesig}(\overline{\alpha}; \Sigma) & \triangleq \{\text{tyvars}\ Y:\{1:[T], \ldots, m:[T]\}, \text{consts} : \{1:S_1', \ldots, n:S_n'\}\}, \quad \text{where } \overline{\alpha} = \alpha_1, \ldots, \alpha_m \text{ and } \Sigma = X_1:S_1, \ldots, X_n:S_n \text{ and } S'_i = S_i[Y, \alpha_1]_{j=1}^n \\
\text{genvars}(\Gamma; \Sigma; \tau) & \triangleq \overline{\alpha}_2, \quad \text{where } \overline{\alpha}_2 \text{ is the greatest set such that } \overline{\alpha}_1 \cup \overline{\alpha}_2 = \text{UV}(\Sigma, \tau) \text{ and } \overline{\alpha}_1 \cap \overline{\alpha}_2 = \emptyset \text{ and } \overline{\alpha}_2 \cap \text{UV}(\Gamma, \Sigma_1) = \emptyset \text{ where } \text{partition}(\overline{\alpha}_2; \Sigma) = (\Sigma_1; \Sigma_2)
\end{align*}
\]

Figure 16. Type Inference Rules for Terms

Figure 17. Auxiliary Definitions for Use in the Polymorphic Generalization Rule
Modules With Constraint and Signature Normalization:

\[ \Theta; \Gamma \vdash mod \Rightarrow M : S/(\Sigma; \delta) \]

\[ \Theta; \Gamma \vdash \text{con} \Rightarrow C : K \]

\[ \Theta; \Gamma \vdash \text{exp} \Rightarrow e : \tau/(\Sigma; \delta) \quad e \text{ not valuable} \]

\[ \Theta; \Gamma \vdash \text{exp} \Rightarrow \nu : \tau/(\Sigma_1; \delta_1) \]

\[ \text{genvars}(\delta_1; \Gamma; \tau) = \tau \quad \text{partitions}(\tau; \Sigma) = (\Sigma_1; \Sigma_2) \]

\[ \alpha = \alpha_1, \ldots, \alpha_m \quad \Sigma_2 = X_1; S_1, \ldots, X_n; S_n \quad \delta = (\alpha_i; \text{tyvars}.i)_{i=1}^n \]

\[ \sigma = \{ X_i \mapsto X.\text{consts}.i \}_{i=1}^n \]

\[ \Theta; \Gamma \vdash \text{mod} \Rightarrow \lambda X.S. [\sigma \delta v] : \forall X.S.[\delta \tau]/(\Sigma_1; \delta_1) \]

Figure 18. Type Inference Rules for Modules
Analogous to corresponding module inference rules, but without any mentions of Σ or δ. The only exceptions to this rule are the ones for canon(s) and \( Mod \Rightarrow S \), which invoke their module-level counterparts directly in the premises:

\[\Theta; \Gamma \vdash canon(s) \Rightarrow M : S/(\ell; id)\]

\[\Theta; \Gamma \vdash Mod \Rightarrow S \Rightarrow M : S/(\ell; id)\]

Calls to coercive signature matching (in various rules) require empty constraint and id substitution in the result.

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

\[\Theta; \Gamma \vdash P \Rightarrow M : S\]

\[\Theta; \Gamma \vdash \text{using } P \text{ in } top \Rightarrow M : S\]

**Figure 21. Type Inference Rules for Top-Level Modules**
We say that an IL kind \( \mathcal{K} \) is **legal** if it has the form of an EL kind.

We say that an IL signature \( \mathcal{S} \) is **legal** if:

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

Essentially, this definition is just trying to ensure that if we (informally) “ran 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 \( \mathcal{C} \) 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 \( \mathcal{S} \) is **synthesis** if all signatures within it are ground, with the following exception: signatures within \( \mathcal{S} \) of the form \( \forall X : R, [\tau] \) may be non-ground so long as they do not appear in the argument of a functor signature.

We say that a context \( \mathcal{C} \) is **synthesis** if \( \forall \alpha : \mathcal{K} \in \mathcal{C} \) is ground and \( \forall X : \mathcal{S} \in \mathcal{C} \) is synthesis.

We write \( \mathcal{C} \vdash \delta \) if all \( \forall \alpha : \mathcal{K} \) in \( \mathcal{C} \) are ground, and \( \forall \alpha \in \text{dom}(\delta) \), \( \mathcal{C} \vdash \delta \alpha : T \).

We write \( \mathcal{C} \vdash \sigma : \Sigma \) to mean that \( \forall X : \Sigma \in \mathcal{C}, \mathcal{C} \vdash \sigma X : \Sigma \).

**Figure 22. Definitions for Soundness**

---

**Theorem (Soundness)**

Suppose \( (\mathcal{C}, \mathcal{C}) \) is valid for inference. For all inference judgments that take \( \mathcal{C}, \mathcal{C} \) as input and return a substitution \( \delta \), we have \( \mathcal{C} \vdash \delta : \mathcal{C} \).

Suppose further that \( \mathcal{C}' \supseteq \mathcal{C}, \mathcal{C}' \vdash \delta' : \delta \mathcal{C}, \vdash (\mathcal{C}', \mathcal{C}') \) ok, and \( \mathcal{C}' \vdash \sigma : \delta' \Sigma \). Then:

1. If \( \mathcal{C} \vdash \text{meta}_1 \equiv \text{meta}_2 \Rightarrow \delta \), then \( \mathcal{C}' \vdash \delta' \text{meta}_1 \equiv \delta' \text{meta}_2 \).
2. If \( \mathcal{C}, \mathcal{C} \vdash \text{knk} \Rightarrow \mathcal{K} \), then \( \mathcal{C}' \vdash \text{knk} \Rightarrow \mathcal{K} \) and \( \mathcal{K} \) is ground.
3. If \( \mathcal{C}, \mathcal{C} \vdash \text{con} \Rightarrow C : \mathcal{K} \), then \( \mathcal{C}' \vdash \text{con} \Rightarrow C : \mathcal{K} \) and \( \mathcal{K} \) and \( \mathcal{C} \) are ground.
4. If \( \mathcal{C}, \mathcal{C} \vdash \text{sig} \Rightarrow S \), then \( \mathcal{C}' \vdash \text{sig} \Rightarrow S \) and \( S \) is ground.
5. If \( \mathcal{C}, \mathcal{C} \vdash \text{exp} \Rightarrow e : \tau / (\Sigma, \delta) \) or \( \mathcal{C}, \mathcal{C} \vdash \text{exp} \Rightarrow e : \tau / (\Sigma, \delta) \), then \( \mathcal{C}' \vdash \text{exp} \Rightarrow \sigma' \delta' e : \delta' \tau \).
6. If \( \mathcal{C}, \mathcal{C} \vdash \text{exp} : \tau \Rightarrow e / (\Sigma, \delta) \), \( \mathcal{C} \vdash \text{exp} : \tau \), and \( \mathcal{C} \) is ground, then \( \mathcal{C}' \vdash \text{exp} \Rightarrow \sigma' \delta' e : \tau \).
7. If \( \mathcal{C}, \mathcal{C} \vdash \text{mod} \Rightarrow M : S / (\Sigma, \delta) \) or \( \mathcal{C}, \mathcal{C} \vdash \text{mod} \Rightarrow M : S / (\Sigma, \delta) \), then \( \mathcal{C}' \vdash \text{mod} \Rightarrow \sigma' \delta' M : \delta' S \).
8. If \( \mathcal{C}, \mathcal{C} \vdash P \preceq S \Rightarrow \mathcal{V} : S / (\Sigma, \delta) \), \( \mathcal{C} \vdash P : S \), \( \mathcal{C} \vdash \text{sig} \), and \( S \) is ground, then \( \mathcal{C}' \vdash \text{mod} \Rightarrow P \preceq S \Rightarrow \delta' \mathcal{V} : S \) and \( S \) is ground.
9. If \( \mathcal{C}, \mathcal{C} \vdash \Sigma_0 \downarrow (\Sigma, \sigma / (\Sigma, \delta)) \) or \( \mathcal{C}, \mathcal{C} \vdash \Sigma_0 \downarrow (\Sigma, \sigma / (\Sigma, \delta)) \), then \( \mathcal{C}' \vdash \text{mod} \Rightarrow \sigma' \delta' \mathcal{V} : \delta' \Sigma_0 \).

Finally: Suppose that \( (\mathcal{C}, \mathcal{C}) \) ok, \( \mathcal{C} \) is ground, and \( \mathcal{C}, \mathcal{C} \vdash \text{top} \Rightarrow M : S \). Then, \( \mathcal{C} \vdash \text{top} \Rightarrow M : S \) and \( S \) is ground.

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