# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>HASKELL AND TYPE CLASSES</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction to Type Classes</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Global Instance Environment</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Inheritance</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Ambiguity</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Background</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Extensions to Type Classes</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Multiparameter Type Classes</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Constructor Classes</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>SML MODULES</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Signatures</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Structures and Functors</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Functor Application</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>Inheritance</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>Equality Types</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>TYPE CLASSES VS MODULES</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Environment</td>
<td>19</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Named Instances</td>
<td>21</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Overlapping Instances</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>MODULES AND TYPE CLASSES</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Modular Type Classes</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Setup</td>
<td>25</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Analysis</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Wehr’s Constructive Comparison</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>Backpack</td>
<td>31</td>
</tr>
<tr>
<td>5.4</td>
<td>Name Spaces in C++</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSION</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>Type Classes in a Strong Module System</td>
<td>36</td>
</tr>
</tbody>
</table>
ABSTRACT

Wadler et al. introduced type classes in Haskell to support a form of generalized ad hoc polymorphism. The equality operator is often overloaded, but it may have interpretations for an infinite class of types including primitive types like integers and strings, and compound types like products and lists [15]. In Standard ML, it is even possible to define functions that are polymorphic over the class of types supporting a computable equality (so called equality types). Type classes were designed to generalize Standard ML’s treatment of overloaded equality, providing a more general solution to the issue of ad hoc polymorphism by extending parametric polymorphism. In parametric polymorphism, functions are independent of type parameters (length of a list, reverse a list, etc). Type classes parameterize over types that have associated interfaces of functions, such as the arithmetic operators for numeric types and the equality operation for types.

In this paper, we provide a critical analysis of the type class system relative to SML modules using examples. We focus on one of the main features of the type class system: the global instance environment [4]. Type inference with type classes relies on a globally-scoped environment that is automatically populated with every instance declared in a program. We discuss the limitations of this approach, including the inability to name instances or restrict their scope. Since type classes rely on the implicit flow of information in Hindley-Milner type inference, they can also create situations in which overloading is hard to mentally resolve when reading code.

Our other main contribution is a discussion of the problem of integrating type classes with a strong module system. One of the limitations of the global instance environment is that it makes integration of type classes with a strong module system infeasible. Wehr presented a constructive comparison of type class and modules with formal translations between the two [20]. Dreyer et al. found Wehr’s translation impractical and proposed an extension to the module system to support type classes [1]. However, Dreyer et al. modify the treatment of

---

1. The instance environment is also referred to as a class environment in some papers [14].
the global instance environment, requiring users to import instances into parts of a program explicitly. Kilpatrick et al. introduce Backpack, a language that goes on top of Haskell’s weak module system to extend its modularity with separately type checked packages [10]. They don’t support type classes yet, and they expect global scoping and lack of named instances to create major challenges. We use ideas and examples from Wehr, Dreyer, and Kilpatrick’s systems to illustrate the problems with a global instance environment in type classes.

Type classes and modules both support a generalized form of parametric polymorphism. Modules have the added benefit of supporting the organization of code into separate, composable units while type classes support implicit typing. Previous attempts to integrate type classes in a ML-like environment have not be successful. In future attempts, it will be beneficial to minimize the overlap of functionality between the two systems, as Fisher and Reppy’s language, MOBY. MOBY integrated objects into a ML-like module system while minimizing their overlap in functionality [3]. A similar approach for type classes and modules would reveal the incompatibilities between the two and whether they can be overcome. Our final contribution is an analysis of the essence of type classes and modules respectively, properties which may interact together more cleanly in an integration of type classes and modules.
CHAPTER 1
INTRODUCTION

We provide a critical analysis of type classes with respect to the SML module system. We use code examples and related language extensions to reveal the limitations of the global instance environment in type classes.

Type classes support an extended form of parametric polymorphism. Arithmetic operators are traditionally overloaded with ad hoc polymorphism, but, in Haskell, ad hoc polymorphism is replaced with a type polymorphism that maps types to interfaces consisting of functions and values (called methods.) A major highlight of the type class system is its implicit typing based on Hindley-Milner type inference. Declared type class instances are automatically added to an instance environment \[4\].\footnote{The instance environment is also referred to as a class environment in some papers \[14\].} This instance environment is global and available in the whole program. However, while the global instance environment is a key concept underlying type classes and their implicit typing, they are incompatible with strong modularity owing to their lack of scoping and naming.

In this paper, we provide a critical evaluation of type classes with respect to the SML module system. The module system was introduced to allow for better code organization and hierarchy; it allows for the declaration of functors, which, when compared to type classes, offer comparable but explicit polymorphism \[11\]. We evaluate snippets of code from Haskell and SML, showing how the global environment violates the modularity expected in a module system.

Finally, we also discuss some of the papers written about the compatibility of type classes with SML modules in three ways: constructive comparison, integration, and extension. In Wehr’s constructive comparison of type classes and modules, he gives a formal translation between type classes and a subset of the module system and vice versa. He uses the translation to compare the two systems and mentions the key differences in the two systems, including namespace management and anonymous signatures \[20\]. His translation
is extremely technical and heavy-handed and doesn’t address the question of whether and how we could integrate type classes with SML modules.

In Modular Type Classes, Dreyer et al. integrate type classes into the SML module system. They propose that type classes are one of the many ways of programming in the SML module system. They extend the module system with a few tricky keywords: `implicit`, `explicit`, `using`, and `overload` to support type class style of extended parametric polymorphism. However, they abandon the global instance environment in Haskell and require users to explicitly import previously declared instances into each program (with the `using` keyword). Their approach is inadequate because naming instances and locally limiting the scope of available instances does not deal with the problem of fully integrating type classes and instances with the module system. Unlike Wehr, they implement a named implicit instance environment, but they do not deal with the problem of passing classes and instances in a functor parameter.

Kilpatrick et al. propose Backpack, a language to build packages on top of the existing weak Haskell module system. Instead of integrating type classes into a brand new, stronger module system, they look to extend the system by building on top of it. Their current implementation supports data types and values; they foresee challenges in resolving class imports due to the global environment in Haskell. Kilpatrick et al. are looking for an expressive practical approach that is compatible with older versions of Haskell, hence they chose a system orthogonal to the existing module system and type classes. While SML has a single module system that handles separation, organization, and code hierarchy along with parametric polymorphism, Haskell now has three (and counting) systems for the same functions, module system, type classes, and package building. This approach leads to a larger, more complex system, even when each component might have a goal of simplicity and practicality.

In this paper, we illustrate the limitations of a global instance environment in type classes and the incompatibility it causes with a strong module system (in this case, the SML module
system). We also illustrate how resolving the environment has created issues in other works that compare or integrate type classes with modules.
CHAPTER 2
HASKELL AND TYPE CLASSES

To motivate type classes in Haskell, let us first talk about how equality is used in programming languages. The following statements are all valid Haskell expressions.

```haskell
>5==6
False
>6.8==4.8
False
>"ha"=="ha"
True
```

The equality operator, `==`, is used to check equality for integers, floats, and strings. Modern languages come with such built-in types and their corresponding implementations of equality. These languages also have a notion of equality for compound types such as lists, tuples, etc.

In Haskell, Wadler and the committee sought a general solution for equality overloading, which led to the invention of type classes [19]. Type classes actually have a greater utility than ad hoc polymorphism; they support an extended form of parametric polymorphism by mapping types over an interface of functions and values. Type classes are ubiquitous in Haskell programming and one of the first features functional programmers learn.

In the example above, `==` is implicitly parametrized over the instances of type class `Eq`. We introduce type classes and their instances using the type class `Eq` as an example. We also discuss some background development in ad hoc and parametric polymorphism. We describe extensions to the original Haskell type class system.
2.1 Introduction to Type Classes

Let us look at a simplified version of the type class declaration of \texttt{Eq}. This is not the definition in \texttt{GHC} which includes more methods, but we will just concentrate on equality checking operator (==) for this example.\footnote{Glasgow Haskell Compiler. The actual \texttt{Eq} class has a method for not equal (\texttt{\neq}) as well as default implementations.}

In Figure 2.1. the class declaration introduces a new type class, \texttt{Eq}, and class methods that should be supported by the type that is an instance of the \texttt{Eq} class. == is a method with type \((\texttt{Eq a})\Rightarrow\texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{Bool}\). For every type \(\texttt{a}\) that is an instance of class \texttt{Eq}, == has type \(\texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{Bool}\). In this case, \texttt{Eq a} is a constraint on the operator == and is called a context \cite{context}. The context specifies that == is only defined for types that are instances of the type class \texttt{Eq}.

For example, \texttt{int} and \texttt{char} are possible instances of the type class \texttt{Eq}. Let us now use the \texttt{Eq} class declaration to create instances of \texttt{Eq} for integers and characters.

\begin{verbatim}
class Eq a where
  == :: a -> a -> Bool
instance Eq Integer where
  x == y = x 'integerEq' y
instance Eq Char where
  x == y = x 'charEq' y
5 == 6
'a'=='a'
\end{verbatim}

Figure 2.1: Instance Declarations of class Eq

Using the same class declaration (\texttt{Eq}), we can create multiple instances of the class with different types. These instances include definitions of the class method ==. We instantiate an equality check between two integers with a built-in function \texttt{integerEq}.\footnote{Actually, a function with this name is not part of the current \texttt{GHC} base, but it lets us make our argument without going into the details of the implementation of integer equality.}

We also use the same type class \texttt{Eq} to create an instance of \texttt{Eq} with instance type \texttt{char}.\footnote{Again, we assume there exists a built-in \texttt{charEq} function that checks if two characters are equal.}

The \texttt{Eq a} class constraint in the type of == means that it expects an \texttt{Eq} instance of the given type in the environment \cite{environment}. Let us look more closely at the environment.
2.1.1 Global Instance Environment

Haskell has a global instance environment that is automatically populated with instance declarations as soon as they appear in a program. By default, the language expects only one instance to unambiguously match the constraint being resolved. In Figure 2.1, Eq Integer would be the required context. If more than one instance matches a class constraint, they are overlapping instances. In the example above, Eq Integer and Eq Char are added to the environment as soon as they appear in a program. This environment is available in the whole program, even if the program consists of multiple modules.

A limitation of the global instance environment is that a type can be declared as a class instance at most once in a program [13]. In some programs, it may make sense to have multiple overlapping instances of the given type in a single program. In the example above, an additional instance declaration of type Eq Integer is illegal. Even today, this is a point of contention in Haskell. Currently, GHC lets programs have more than one instance match with flag -XOverlappingInstances, but the behavior of type inference is not well defined in this situation [18].

A benefit of the global instance environment is that users don’t have to explicitly specify which instance declaration they want used in a program; the environment remembers exactly one instance declaration for a given type. In the example, 5==6, we don’t have to specify that we wish to use the method in the Eq Integer declaration. Since there is only one allowed instance of the == operator with type Int -> Int -> Bool, the type checker uses that instance method in the expression.

The global instance environment is an essential part of the type inference algorithm in Haskell. While inferring the type of a variable, the inference algorithm checks whether a method is a member of a defined type class. The global instance environment ensures that a method name is unique in a class. For example, we cannot have a method eq that exists in two different class definitions. The inference algorithm also uses the fact that a class name and instance type uniquely determine a class instance. A global environment greatly
simplifies ensuring this uniqueness. With modular instance environments, overlapping of method names or class instance types would have to be resolved at multiple points of the program. The design considerations that arise are complex and would need to be carefully worked out in a language that integrates type classes in a strong modular system.

We will give more details of the global instance environment and its implications in chapter 4 in context of the SML module system.

### 2.1.2 Inheritance

Let’s take a look at another class, `LessEq`, which allows you to compare two parameters with the `<=` operator. Logically, separate `Eq` and `Less` classes are not prerequisites to define `LessEq`, but we will use this example to demonstrate inheritance in Haskell.

```haskell
class Less a where
  == :: a -> a -> Bool

instance Less Integer where
  x < y = x `<integerLess'> y

class (Eq a, Less a) => (LessEq a) where
  (<=) :: a -> a -> Bool

instance LessEq Integer where
  x <= y = x `==` y || x < y
```

Figure 2.2: Class and Instance Declarations of Less

In an implementation of `LessEq`, we assume that the global instance environment has an implementation of `Eq` that provides us the `==` method. We also require an instance of `Less` that will provide the `<` method.

The type of `<=` is `(LessEq a) => a -> a -> Bool`. Notice, `LessEq` has `Eq` and `Less` as class constraints, or, conversely `Less` and `Eq` are superclasses of `LessEq` [18].
2.1.3 Ambiguity

In order to talk about ambiguity in Haskell, let us look at the class definition of `Read`.

```haskell
class Read a where
    val read :: String -> a
```

The `Read` class converts strings into arbitrary types that are instances of `Read`. The type of the `read` method is `(Read a) => String -> a`. Some examples of expressions that contain `read` are below:

1. `read("5") - 10`
2. `read("5.5") - 6.0`
3. `read("5.5")`
4. `read("5.5") - 6`
5. `read("5.5")::float`

From the expression, the read type returns an integer in `read("5") - 10` and a float in `read("5.5") - 6.0`.

The expression `read("5.5") - 6` throws an error as the `read` function attempts to coerce a float `5.5` expressed as a string into a integer. The error is *** Exception: Prelude.read: no parse.

In the expression `read("5.5")`, the type of the expression is ambiguous since it has no specified output type. The read type class requires a specific type it needs to return for it to type check. We could explicitly annotate a type to a read expression to avoid this error, as in `read("5.5")::float`.

The `Read` class demonstrates possible ambiguity in Haskell due to implicit typing.
2.2 Background

Type classes introduced generalized ad hoc polymorphism in Haskell. The design committee were looking for a form of polymorphism beyond ad hoc overloading. In Haskell, functions support a brand of parametric polymorphism, even if the implementation of the function depends on the given type. Type classes represent collections of types that have associated interfaces of functions.

With type classes, functions and operators have a bounded polymorphic quantifier. Instead of ranging over all types in Haskell, they range over a possibly infinite class of types for corresponding instances of Eq. In figure 2.1, the type of \( \text{==} \) is \((\text{Eq } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}\). The bounded polymorphic quantifier or context is to the left-hand side of the \( \Rightarrow \) arrow; in this case, the context is \( \text{Eq } A \). Only types that are instances of the class \( \text{Eq} \) can type check with the method \( \text{==} \), i.e. \text{Integer} \ and \text{Char}. If we declare an instance of \( \text{Eq} \) over lists, the methods in class \( \text{Eq} \) are quantified over an infinite set of types: \text{Integer List, Char List, Integer List List, Char List List}, etc.

The initial type class design was experimental and primitive. With its widespread use, its shortcomings were highlighted. Jones [7], Peyton-Jones [9], and others have developed and integrated extensions to the preliminary type system to meet these shortcomings. Peyton-Jones gives a thorough overview of the design considerations while extending the type system [9]. We will review the major extensions to Haskell type classes.

2.3 Extensions to Type Classes

2.3.1 Multiparameter Type Classes

One of first and most requested extensions to type classes; multiple parameters are now supported in GHC (with the flag \(-XMultiParamTypeClasses\)), though not by default. Multiparameter classes are a way to express the relation between types. They are necessary when a specific instance declaration of a class depends on a tuple of types.
Mathematical collections are a good example to motivate multiparameter classes. Most collections have a set of common functions (empty?, union, intersection, remove, insert, etc,) but the implementation of each function varies depending on the collection as well the type of its elements. For instance, to lookup an element in a set, we need the collection type (set, bag, etc) and the element type (Int, Bool, etc.) In a single parameter type class world, a `Collection` class declaration would look like this:

```haskell
class Collection c where
    lookup:: a -> c a -> Bool
```

In the above example, `c` is the type constructor that forms the collection type `c a`. The type `a` is universally quantified; it is not specific to this function. We might need to test equality between two elements in a lookup function, but we can’t do that with a universally quantified type `a`. With multiparametricity, we could have a class declaration with two type parameters, `c`, the collection constructor, and `a`, the element type.

```haskell
class Collection c a where
    lookup:: a -> c a -> Bool
```

The type of `a` is scoped in each instance declaration. We can enforce restrictions as needed on `a` and apply equality or comparison checks on `a`.

Functional dependencies are available in GHC, but not as a default. Functional dependencies are used to solve ambiguity issues that arise because of multiple parameters [8]. They allow us to express the relationship between multiple type parameters. In a collection, instead of specifying the type constructor of a collection type as `c`, we can define the type of the collection with `ca`. Now let us look to redefine the `Collection` class with the type of the collection:
class Collection a ca | ca -> a where
    lookup:: a -> ca -> Bool

With an annotation \( \text{ca} \rightarrow \text{a} \) to the class declaration of Collection, we declare that \( \text{ca} \) uniquely defines \( \text{a} \) \[8\]. For example, a List definition of the Collection class may have only one class definition of a List of type \( \text{a} \). Due to functional dependencies, since \( \text{[a]} \) uniquely defines \( \text{[a]} \), the following Collection class definitions would be illegal:

Given:

class Collection a [a] | [a] -> a where...

Illegal:

class Collection b [a] | [a] -> b where...

2.3.2 Constructor Classes

Jones \[7\] proposed constructor classes to allow an interface to be attached to an n-ary (\( \star \rightarrow \star \), \( \star \rightarrow \star \rightarrow \star \), etc) type constructors, such as the monad type. With constructor classes, which are now part of Haskell, a type class can contain instances of type constructors along with the implementations of the class interface \[6, 7\].

Let us look at the implementation of functors in GHC:

data Tree a = Leaf a | Branch (Tree a) (Tree a)

class Functor func where
    fmap :: (a -> b) -> (func a -> func b)

instance Functor [] where
instance Functor Tree where
    fmap fun (Branch tl tr) = Branch (fmap fun tl) (fmap fun tr)
    fmap fun (Leaf t) = Leaf (fun t)

The Functor class can now be instantiated with trees, lists, or other abstract data types, as you see above. You’ll notice that in the case of lists, an fmap is simply a map. However, in the case of trees, we define how to structurally decompose the tree to apply a function to every leaf.

We can automatically derive class instances of a datatype. Derived instances are supported for classes in the standard library (Eq, Ord, etc.) We can also attach the deriving keyword to the datatype definition [18, 13].

data Tree a = Leaf a | Branch (Tree a) (Tree a)
deriving (Eq, Ord, Read, Show)

OR
deriving instance Eq a => Eq Tree a

The compiler creates instance declarations such as Eq Tree, Ord Tree, Read Tree, and Show Tree.
CHAPTER 3
SML MODULES

3.1 Introduction

Modules in SML are expressive constructs that contribute to the creation of concise and reusable code. Another application of modules is to extend the polymorphism in Core ML.

Core ML supports parametric polymorphism, in which definitions of functions have uniform implementations regardless of type. An example of parametric polymorphism is length of a list; the function can be instantiated for a list of any given type, but its implementation is unaffected by the given type [11]. Modules support a generalized form of polymorphism that extends parametric polymorphism to functions such as sort, which were impossible to define uniformly for all types. With parametrized modules, we can define sort over elements of a structure that has an associated ordering (i.e. int, char, etc). A module signature specifies the type of a structure.

The module system in SML can be divided into signatures, structures/functors, and functor application. These characterize specification, abstraction, and instantiation respectively.

3.2 Signatures

Signatures are type specifications for an environment. Let us create the signatures EQ that will support equality for types.

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

The signature for EQ can specify abstractions and implementations of EQ. Each instance of EQ will contain an implementation of the eq function.
3.3 Structures and Functors

In SML, the separation of declaration environments from usage environments is through functors and functor applications. A functor allows users to create abstractions of their code and is useful for a powerful explicit form of parametric polymorphism. It allows us to parametrize over signatures that specify types and associated interface values.

We can create multiple abstractions of the EQ module. If we were to create module with this signature (say for int), let us call it EInt.

```sml
structure EInt : EQ =
struct
  type elem = int
  val eq = (op =)
end

functor EList(structure X:EQ) : EQ =
struct
  type elem = X.elem list
  fun eq(x::xlist,y::ylist) = X.eq(x,y) andalso eq(xlist,ylist)
  | eq([],[]) = true
  | eq (_,_) = false
end
```

We can also create an abstraction over lists as well. X represents the structure of type EQ and provides the type specifications for elem and eq in the functor EList.

3.4 Functor Application

After abstraction, we can insert an instance of a module into a program with functor application as shown below:
structure E = EInt();
E.eq(5,6);
>val it = false : bool
structure El = EList(structure X = E);
El.eq([1],[1]);
>val it = true : bool

Now we can use the the `eq` function in the module `E.eq(5,6)` as long as the declaration of `E` is within the current scope.

### 3.5 Inheritance

Parameterization and inheritance are important design considerations while formalizing and implementing modules. However they are complex to implement, especially when supporting parameterization within inheritance [11].

Modules allow for the hierarchical organization of code through inheritance. We use a slightly different example than the one we covered in `EQ` because of the more complex issues that arise when we try to implement Haskell code in the previous chapter in SML. We will cover that example in the next chapter.

We can declare a signature for the module `ORD` that has less than and greater than functions. We can then apply the `ORD` function to check order in lists in SML in `SORT`. In this mini-implementation, we write an `asc` function that checks whether the elements in a list are in ascending order, an order which is defined in an `ORD` functor.

```ml
signature ORD = sig
  type elem
  val less : elem * elem -> bool
  and greater: elem * elem -> bool
end
```
signature SORT = sig
structure O:ORD
val asc: O.elem list -> bool
end

With signatures of ORD and SORT, we then create structures for each signature. To demonstrate inheritance, we provide the ORD structure as a parameter in the SORT functor. In SML, we have to explicitly provide this instance, as we do in SortIntList(structure O = OrdInt()).

structure OrdInt : ORD = struct
type elem = int
val less = (op <)
and greater = (op >)
end

functor Sort(O:ORD): SORT = struct
structure O:ORD = O
fun asc(x:: y:: ylist) = O.less(x,y) andalso asc(y::ylist)
   | asc([]) = true
   | asc(x::nil) = true
end

After this explicit statement, we can then call the asc function on an int list to get a boolean result.

structure SIList = Sort(OrdInt());
SIList.asc([2,3,4]);
In chapter 4, we'll cover environments, instance declaration scope, and type hierarchy in SML in more detail, with respect to type classes in Haskell.

### 3.6 Equality Types

In SML, equality types are mostly independent of the module system. The language introduces an ad hoc fix for equality checking; it has a type quantification that ranges over the equality types (for which built-in equality is defined.) These types are denoted by \( \forall \) type variables. Equality is defined structurally for compound types such as lists, trees, tuples, etc. Additionally, in SML, equality does not apply to functions and reals and abstract types

\[
\text{val } \varepsilon : \forall a \times \forall a \rightarrow \text{bool}
\]

SML generates specialized code for the given type. However, if we introduce a new abstract type, even if we know how to calculate equality for this type, we couldn’t introduce it into the generic built-in equality collection. This is a disadvantage in SML, since the system lacks uniformity in the treatment of abstract versus user-defined types.
CHAPTER 4
TYPE CLASSES VS MODULES

Arithmetic operators, such as addition and multiplication, are typically overloaded, with different interpretations of the operations being chosen based on the types of their arguments [15]. The equality operator is also often overloaded, but it may have interpretations for an infinite class of types including primitive types like integers and strings, and compound types like products and lists. In Standard ML, it is possible to define functions that are polymorphic over the class of types supporting a computable equality (so called equality types). Type classes were designed to generalize Standard ML’s treatment of overloaded equality, providing a more general solution to the issue of ad hoc polymorphism by extending parametric polymorphism. In parametric polymorphism, functions are independent of type parameters (length of a list, reverse a list, etc). Type classes parameterize over types that have associated interfaces of functions, such as the arithmetic operators for numeric types and the equality operation for types.

Modules allow for a generalized form of parametric polymorphism, with functor abstraction allowing for polymorphism over data types and structures, but there are key differences between type classes and modules. We will provide a critical analysis of type classes relative to modules using examples and type theory.

There has been extensive work done to explore how to integrate type classes and modules. Attempts have also been made to define the relationship between the two, formally and practically. In this chapter, we will critique the major limitations of type class design that make it incompatible with an ML-like module system.

Since the functionality of type classes and modules overlap, in this chapter, we will critique the shortcomings of type classes with corresponding code in Standard ML. We will highlight the global instance environment in Haskell, the inability to name instances, and complex implicit overloading.
4.1 Environment

Type classes have a global instance environment: a mapping of class names and the types of their declared instances. Every instance declaration is automatically added to this environment and is available for use throughout the program. When the type inference algorithm encounters an overloaded method, it looks for an instance that matches the constraint it is trying to resolve. Once we have the instance, we lookup the overloaded symbol name in its dictionary to obtain the actual method to call at that point. In the case of the member method function for lists, the \(==\) operators context involves an implicit polymorphically bound type variable constrained by a context (in this case \(\text{Eq} \ a\)), and in this situation the interpretation of the overloaded symbol is given by a selection from an (implicit) dynamically passed dictionary.

We defined classes \textbf{Less} and \textbf{Eq} and instantiated them for integers in Figures 2.1 and 2.1.2. We used \textbf{Eq} and \textbf{Less} as class constraints to declare the \textbf{LessEq} class and instantiated it for type \textbf{Integer}. Let us look at the definition of \textbf{LessEq} again in figure 4.1.

\begin{verbatim}
class (Eq a, Less a) => (LessEq a) where
  (<=) :: a -> a -> Bool

instance LessEq Integer where
  x <= y = x == y || x < y

4 <= 5;
\end{verbatim}

Figure 4.1: Class and Instance Declarations of Less

The type system allows us to define class constraints for each type in a type class. For example, the type \((\text{LessEq} \ a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}\) requires \(a\) to be a member of class \textbf{LessEq}. In turn, from the class declaration, \(a\) is expected to be a member of \textbf{Less} and \textbf{Eq}.

We recreate the \textbf{Eq}, \textbf{Less}, and \textbf{LessEq} examples in SML. We declare signatures for \textbf{EQ} and \textbf{LESS}. Since SML does not allow user-defined operator overloading, we use function names \textbf{eq} and \textbf{less} instead of \(=\) and \(<\) respectively.

The \textbf{eq} function has type \textbf{elem*elem -> bool}. In Haskell, the type of \(==\) was \((\text{Eq} \ a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}\). SML does not support class constraints in the type of a function, but
signature EQ = sig
  type elem
  val eq : elem * elem -> bool
end

signature LESS = sig
  type elem
  val less : elem * elem -> bool
end

structure IntEq : EQ = struct
  type elem = int
  val eq = (op =)
end

structure IntLess : LESS = struct
  type elem = int
  val less = (op <)
end

Figure 4.2: Class and Instance Declarations of Less

when eq is used in an expression in a program, we have to refer to a specific named structure with signature EQ (i.e. IntEq.eq(2,3)).

Next, we will declare structures with signature EQ and LESS with type parameter int. Since SML allows naming of structures and functors, we call the instances IntEq and IntLess respectively.

Let us now look at the signature of LESSEQ, which corresponds to the LessEq type class in Figure 4.1. In Haskell, we can specify class constraints for the <= method with (Eq a, Less a) => LessEq a. SML, on the other hand, requires us to name the two expected instances of EQ and LESS, and specify which type they share (elem) using the sharing keyword.

signature LESSEQ = sig
  structure E:EQ
  structure L:LESS
  sharing type E.elem = L.elem
  type elem = E.elem
  val le: elem * elem -> bool
end

Figure 4.3: Class and Instance Declarations of Less

We need to specify the structure or functor by name, since there can be multiple instances
of the same type but different names in a program.

```sml
functor LessEqual (E':EQ; L':LESS;  
sharing type E'.elem = L'.elem): LESSEQ = struct  
  structure E:EQ = E'  
  structure L:LESS = L'  
  type elem = E.elem  
  fun le (m,n) = L.less(m,n) orelse E.eq(m,n)
end

structure IntLessEq = LessEqual(E' = IntEq; L' = IntLess);  
structure IntLessEqCopy = LessEqual(E' = IntEq; L' = IntLess);
```

Figure 4.4: Class and Instance Declarations of Less

Now we can create a functor abstraction over the LESSEQ signature for integers. In this case, we specify that we expect an EQ and LESS structure. We can then go on to define the function le with respect to L and E, using the eq function in EQ and the less function in LESS.

We perform functor application in LessEqual, providing IntEq and IntLess as parameters to correspond to the required EQ and LESS structures. Since SML allows us to name structures, we can create identical structures with different names, IntLessEq and IntLessEqCopy.

This example demonstrates overlap in functionality in modules and type classes. SML modules are explicit and can be cumbersome to read or write for smaller examples. However, we will look at more complicated examples in which the explicit functor parameters are easier to read and understand over than the implicit overloading of Haskell.

### 4.1.1 Named Instances

Named instances are not supported in Haskell. The module system in SML expects users to name structures or functors, except in the case of anonymous structures. In Haskell, an instance is accessible in the type checker via its class name and the instance type. There is no support to give instances custom names in the semantics. For example, the class instance `Eq Integer` is indexed by its class name, Eq, and its type parameter, Integer.
In SML, we can define a unique name for the structure, even when it is a result of a functor application. For example, in Figure 4.1, \texttt{IntLessEq} is a structure. If we want to use that structure later in the program, we have to refer to the function or type in the structure. We can use the function \texttt{le} in a program with the expression \texttt{IntLessEq.le(5,6)}. We can also create identical structures with different names, \texttt{IntLessEq} and \texttt{IntLessEqCopy}.

More examples of named instances can seen earlier in this chapter in the Figure 4.1 in the instances \texttt{IntEq} and \texttt{IntLess}.

\subsection*{4.1.2 Overlapping Instances}

In the Haskell instance environment, there can only be one instance at a given class of a given type. If in a given program, there are multiple instances that match the same class constraint, we have a case of overlapping instances, and the type inference fails [18]. The problem of overlapping instances is due to two main shortcomings of type classes: the global instance environment and unnamed instances.

Class instances cannot be named. Instead of indexing the instance environment by name, it is indexed on the class name and the instance type. Since the instance environment is global, there can be only one class of a given type.

SML supports scoping. It has a modular hierarchical structure, there can be multiple names for the same type. The user explicitly specifies a particular module to use in different parts of the code. The "open" keyword allows users to preserve modularity by importing a previously defined module into the current environment.

For instance, if \texttt{eq} were implemented in the \texttt{IntEq} module, there are two ways the user can reference the \texttt{EQ} class. We can either specify the module (\texttt{IntEq}) and the requested function (\texttt{eq}), or if we plan make multiple calls to functions in \texttt{IntEq}, \texttt{open IntEq} brings all the functions in it into the current environment, and we can then reference them directly.

\texttt{IntEq.eq(3,4)}
In Figure 4.1.2, in Haskell, we write two instances of a Functor over a tree that are mirror images of each other.

```haskell
data Tree a = Branch (Tree a) (Tree a) | Leaf a

instance Functor Tree where
  fmap fun (Branch tl tr) = Branch (fmap fun tl) (fmap fun tr)
  fmap fun (Leaf t) = Leaf (fun t)

instance Functor Tree where
  fmap fun (Branch tl tr) = Branch (fmap fun tr) (fmap fun tl)
  fmap fun (Leaf t) = Leaf (fun t)
```

Figure 4.5: Class and Instance Declarations of Less

This example demonstrates that Haskell does not accept overlapping instances in the same file. The second Functor creates a mirror image of a tree. When we type check this file, we get the following error:

```
Duplicate instance declarations:
  instance Functor Tree -- Defined at code_tc.hs:......
  instance Functor Tree -- Defined at code_tc.hs:......
Failed, modules loaded: none.
```

The type class context types of Haskell can be recreated explicitly in SML, if the user can specify each of the type instances in the context. Look at `IntLessEqual`, that corresponds to the function of type `(LessEq a) => a -> a -> bool` in Haskell. To create a usable structure in code, we have to provide the compiler with instances of `EQ` and `LESS` that corresponds to the type of `IntLessEqual` (in this case, `int`). So, `(E':EQ, L':LESS)` explicitly refer to the same functionality the Haskell type inference engine implicitly provides.
4.2 Overview

We have detailed the major differences between type classes and modules. This lays the foundation for discussing work to combine the features of the two systems. In the next chapter, we’ll talk about Dreyer’s Modular Type Class [1], Kilpatrick’s Backpack language [10], and Wehr’s constructive comparison [20].
CHAPTER 5
MODULES AND TYPE CLASSES

Of the many attempts to combine the benefits of type classes and modules, Dreyer’s Modular Type Classes is one of the recent practical and theoretical efforts [1]. Wehr also provides a constructive comparison with a translation from type classes to subset of modules and vice versa [20]. Recent interesting work also includes Backpack, a package management language that goes on top of the Haskell module and type system [10].

5.1 Modular Type Classes

Modules allow us to break large programs into self-contained components. Modules and type classes have some overlap in their functionality, as we mentioned in previous chapters. They both allow us to provide an interpretation of a type. We explore whether type classes are compatible with a strong module system (unlike the weak one in Haskell.)

Dreyer et al. treat type classes as a mode of programming with modules. They start with modules, add a few keywords (implicit, explicit, etc) to recreate type classes in a strong module system. In this section, we will use Dreyer’s approach to use a SML-style language to recreate the functionality of type classes. In order to highlight the differences between SML, Haskell, and Dreyer’s system, we will compare the same example implemented in each of the three languages side-by-side.

5.1.1 Setup

In Dreyer’s system, type class declarations in Haskell are represented with module signatures in SML.

In Haskell

class Eq a where

In SML

signature Eq =

end
In SML
signature EQ =
sig
  type elem
  val eq : elem * elem -> bool
end

In MTC
signature EQ =
sig
  type elem
  val eq : elem * elem -> bool
end

If you compare the SML and MTC examples above, you see that they are identical in this case. More differences will arise in the MTC vs Haskell. You notice that instead of a function called eq, Haskell lets you override the \( \Rightarrow \) symbol for user defined types. Such operator overloading is impossible in SML.

In MTC, type class instances correspond to structures and functors in the module system.

In Haskell,
instance Eq Int where
  \( \Rightarrow \) = integerEq
deriving instance (Eq a, Eq b) => Eq(a,b)
OR
instance (Eq a, Eq b) => Eq (a,b) where
  \((x1,y1)\Rightarrow(x2,y2)\) = (x1\Rightarrow x2) && (y1\Rightarrow y2)
In SML
structure EqInt: EQ = struct
  type a = int
  val eq = Int.=
functor EProd(X:EQ; Y:EQ) = struct
  type elem = X.elem * Y.elem
  fun eq((x1,y1), (x2,y2)) = X.eq(x1,x2) andalso Y.eq(y1,y2)
end

In MTC
structure EqInt: EQ = struct
  type a = int
  val == = Int.=
end

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

Deriving is a notational way of automatically creating instances of classes in the standard library in Haskell. We can also create derived instances of user-defined type classes if the inference algorithm can determine what the class instance will look like. While we are unable to name class instances in Haskell, MTC allow us to name class instances, a crucial difference in the two systems. This ability to have a named instance allows us to explicitly import a specific instance, include a given instance in a signature, and allow multiple class instances with the same instance type with different names. We will go into more detail in section
5.1.2.

Type classes have a global environment that is populated on class instantiation, while modules have explicit scope. In SML, modules specifically scope and create hierarchy in a program while the flat type class instance global environment allows for implicit type inference. In Haskell, Dreyer has a global environment of the names of class instances, but requires the user to explicitly import classes into the top-level of a module with **using**. It looks similar to the **use** mechanism in SML/NJ, except instead of importing files, it imports instances to a module. However, we may have to explicitly import numerous instances with the **using** keyword in moderate to large programming projects, and the notation is cumbersome if we need to import more than handful of instances. A way to specify instances in a signature and import modules would be ideal.

**In Haskell**

**Global Instance Environment**

**In SML**

```
open <structure name>
```

**In MTC**

```
using EqProd in mod
```

While Haskell doesn’t allow named instances, SML supports named and anonymous structures. Dreyer et al. use the **overload** keyword to bind names to methods in classes. The explicit import of class instances can also cause some overlap with existing classes, following rules that they detail in their implementation. There are also **implicit** and **explicit** keywords in their extension to SML, to coerce explicit to implicit constructors and vice versa. In MTC, users have the option of working in the Haskell implicit type inference or in the SML explicit types with these keywords.

28
Haskell

No Named Instances

In SML

structure EqIntProd = EqProd(EqInt, EqInt);

In MTC

val eq = overload eq from EQ

5.1.2 Analysis

Dreyer et al. attempt to integrate type classes into a strong module system. They treat type classes as a mode of programming in modules, and use the Harper-Stone interpretation of Standard ML in the translation of type classes in the SML module system [1, 5].

A clean language design separates core language elements (datatypes, values, types, and other details) from module level constructs (the layer of language in which the core language lives.) However, since Dreyer et al. model type classes with modules, they blur the difference between the core and module language. For instance, an Eq type class can be instantiated over a List, a datatype which is part of the core language. The Monad type class interfaces directly with the Monad constructor which is parameterizing over type constructors at the module level [12]. However, Deryer et al. consider constructor classes to be an orthogonal extension to type classes and since modules are higher-order, don’t work out how their system behaves when we have abstractions in functors.

1) Mapping classes to modules may or may not be the only approach. If type classes are part of the core language, we need the ability to specify type classes and instances in a module signature. In other words, we need the ability to name instances.

2) There is overlap between the functionality of type classes and modules. We have described similar examples all through this chapter and chapter 4, using type classes in
Haskell and modules in SML. A clean integration of type classes in a strong module system would involve minimizing this overlap. We have more details in section 6.1.

Another important characteristic of the type class system is its global instance environment. Dreyer et al. mimic this environment to an extent, but require users to explicitly import class instances at the top level of a program. While the users can control the scope of instances, all the declared instances are in a global environment that is available at the top level. It follows that in two different modules, we couldn’t have type class instances with the same name. Dreyer et al. give an impression of scoping by requiring an explicit import statement, but their global instance environment is still flat, with every declared instance importable at the top level of each program.

A final critique is that Dreyer et al. only handle the initial type class design, not extensions such as multiparameter classes and constructor classes. They treat the constructor class extension as orthogonal to the original type class design, but higher order modules will further complicate the language and blur the distinction between the core and module language. Integrating further extensions to their mapping of type classes to module will create a heavy-weight system that doesn’t preserve the simplicity of Standard ML.

Hence, while Dreyer et al. provide a mapping of type classes to modules in SML, questions still remain. 1) Are type classes a core or module level construct?

2) Should type classes be stripped down to minimize overlap with module functionality?

3) What is the scope of the type class instance? How do we attach names and modularity to a type class instance?

### 5.2 Wehr’s Constructive Comparison

In his Master’s thesis, Wehr provides a translation of type classes to modules and vice versa. However, his translation is too heavyweight to suggest a clean language that has type classes and ML-style modules. His translation doesn’t show us how to practically integrate type classes with a stronger module system.
Wehr’s work shows that there exists a theoretical translations between type classes and a modules. He uses the theoretical translations to point out apparent differences between type classes and modules. As we do, he mentions instance scoping, named instances, and overloading, with a short 2-3 sentence description each. We have expanded on those and other points of differentiation, providing examples as well insights into language design. We use these differences to evaluate properties of system with a strong module system and type classes.

5.3 Backpack

In Backpack, Kilpatrick et al. approach the problem of integrating type classes with modules from a different perspective. Instead of modifying Haskell’s weak module system, they create a language to build packages on top of a weak module system (and type classes). Instead of looking for theoretical simplicity, their goal is to create a lightweight practical solution to modularity in Haskell while preserving the existing system of modules, type classes, and Cabal (a package manager, similar to the CM compilation manager in Standard ML) [18].

Backpack provides an explicit interface with recursive linking. It is based on previous work in MixML, but instead of recreating all modular functions in Backpack, Kilpatrick looks to build on the existing module system in Haskell. So far, they support data types and values. They are yet to publish work on representing type class definitions and instances in Backpack.

An example of a package in Backpack is below. Package abcd-holes has two holes (or placeholders) for modules A and B. The type of abcd-holes provides four modules: C (which provides \( x :: \text{Bool} \)), D (which provides \( z :: \text{Bool} \)), A (which provides \( x :: \text{Bool} \)), and B (which provides \( y :: \text{Bool} \)). The typed holes A and B allow them to type check C and D even without the import of A or B.

package abcd-holes where
A :: [x:: Bool]
B :: [y::Bool]
C = [x = False]
D = [import qualified A
import qualified C
z = A.x && C.x ]

They foresee problems in scoping class instances due to the global instance environment expected in type classes. They expect to solve this problem with naming instances in their internal language while resolving scope. We look forward to future publications from them.

While this is a practical approach to stay compatible with current Haskell code, Haskell now has multiple extensions: a module system, type classes, Backpack, Cabal, etc. A cleaner language design is to create a start with a core language and module language, and smoothly integrate type classes into its core language.

### 5.4 Name Spaces in C++

C++ is a systems programming language with light-weight abstraction [17]. It is a large programming language that includes features such as templates, classes, and name spaces. Stroustrup’s design for name spaces is to facilitate library design and to scope objects, variables, and functions declared in such libraries [16]. While these design constructs are in the context of objects, C++ also has some functional elements. His approach to resolve scoping in C++ is worth critiquing to address the problematic global instance environment in type classes.

Suppose we have two libraries in C++, first.h and second.h, each with a function f.

```
//first.h:
char f(char);
```

....
//second.h:
char f(char);

Linking the two libraries may cause name clashes, similar to the overlapping classes problem in Haskell. Stroustrup was looking for a way to disambiguate between the variable \texttt{f} in both libraries. Specifically, he was looking for a solution that library code providers could compile into the object code. Since C++ is a large language, it had multiple workarounds including renaming the variable, macro hacking, and wrapping the variable in an object.

Name spaces allow us to declare a scope for a set of expressions in what was traditionally a global environment. We can then explicitly refer to the name space to access an object in it or even open the name space to make all its members visible. Let us look at an example of name space \texttt{A}.

```cpp
namespace A {
    void f(int);
    void f(char);
    class String {...};
    ....
}
```

The class \texttt{String} in \texttt{A} does not overlap with the global declaration for \texttt{String}. In this version of \texttt{String}, we could declare case-independent string comparison or an alternative representation of its contents.

Now, if we were to use this specific class \texttt{String} in another file, we could explicitly qualify it or import it into scope with a \texttt{using} declaration.

```cpp
A:: String s;
OR
using A::String;
String s;
```
In both these declarations, the type of \texttt{s} is the class \texttt{String} declared in name space \texttt{A}. After the \texttt{using} declaration, any other operations on \texttt{s} will default to the those declared in name space \texttt{A}.

Let us now look at an example from Stroustrup’s paper that introduces how overloading and name space management are integrated in C++ [16].

```cpp
namespace X {
    int i, j, k;
}
int k;
void f1()
{
    int i = 0;
    using namespace X; // make names from X accessible
    i++; // local i
    j++; // X::j
    k++; // error: X::k or global k ?
}
void f2()
{
    int i = 0;
    using X::i; // error: i declared twice in f2()
    using X::j;
    using X::k; // hides global k
    i++;
    j++; // X::j
    k++; // X::k
}
```
In function $f_1$, we make the names in name space $X$ accessible, but the resultant scope is
confusing because of multiple rules for overloading:

1) Variable $i$ is resolved to a local instance instead of $X::i$
2) Variable $j$ is resolved to $X::j$
3) Variable $k$ is ambiguous with two declarations in scope: the gobal $k$ and the $k$ in the
name space $X$

In function $f_2$, there are even more ways of handling overloading resolution. The expres-
sion $k++$ is resolved to $X::k$ by default even with a global declaration of $k$.

If we were to to use the ideas from namespace management and overloading resolution to
integrate type classes into a strong module system, we would need a more consistent way to
handle overloading, which will be further complicated with functor abstraction, type class
instantiation, and abstract types. But Stroustrup’s detailed description of design considera-
tions for namespace management with overloading resolution would be worth exploring for
approaches that might transfer over to a formal ML-like module system with type classes
[16].
CHAPTER 6
CONCLUSION

6.1 Type Classes in a Strong Module System

So far we have provided a critical analysis of type classes in the context of a strong module system such as the SML modules. The overlap of functionality between type classes and modules and the global instance environment in type classes cause complications in any integration of type classes with SML modules. What would be the criteria for language design that accommodates type classes and modules?

Fisher and Reppy have related work in object-oriented programming to reduce the overlap between objects and modules in MOBY [2, 3]. They tackle a similar problem in the object-oriented world: combining the strong class mechanisms of Java/C++ with a strong ML-like module system. They allow the module system to provide signatures and parametrization while using classes to support inheritance. A similar solution with type classes and ML-like modules would be ideal.

A global environment while a core feature of the Haskell type system, isn’t modular since the environment is available regardless of scope and location in a program. The central nature of the environment lends itself nicely to implicit type checking in Haskell.

A environment in SML would additionally need to have ability to support scope and named instances.

Is a central table is essential to replicate the utility of type classes? Would another more modular structure work in the SML system?

These problems remain unsolved, although Modular Type Classes was a useful first step toward a solution. Whether this attempt can be carried forward to a complete solution is unclear.
REFERENCES


