Extracting the Essence of Type Classes

David Darais
University of Utah
Bachelor's Thesis
david.darais@utah.edu

May 13, 2010

Abstract

Type classes have been established as an extremely valuable abstraction tool in programming. Initially conceived to support reliable operator overloading, type classes are also capable of expressing complex relationships between abstract concepts. In languages that support them, type classes are closely coupled with the existence of a type system, and in particular its ability to infer types. This paper shows how we can remove the need for a type system from type classes. Without the concept of types, *aspects* and *aspect classes* emerge as the analog of types and type classes. *Aspect classes* lack the convenience of type inference, but the liberation from a type system allows for a more general and expressive semantic dispatch system. This paper attempts to justify the usefulness type classes, explores typeless aspect classes, and provides an implementation of multi-parameter aspect classes and aspect families in PLT Scheme.

1 Motivation For Type Classes

1.1 Background

Before exploring the motivation for implementing type classes in programming languages that currently do not support them, a general understanding of type classes and some of their crucial applications must first be achieved.

1.1.1 Type Classes

The feature set of type classes is twofold; they allow for both the specification of common interfaces and for a method of compile time dispatch in the use of these interfaces. Much in the same way that an object oriented class system allows for dispatch on a common interface by the lookup of a *method* from an *object* at *runtime*, type classes allow for dispatch on a common interface by the lookup of a *function* based on a *type* at *compile time*[1].

Two simple type classes in Haskell are Show and Read

```
class Show a where
  show :: a -> String
class Read a where
  read :: String -> a
```

where the Show class expresses the ability for values of a given type to be represented as a String, and the Read class expresses the ability for values of a given type to be created from a String representation. Instances[2] of Show and Read for the Bool type may be written as

```
instance Show Bool where
  show True  = "yes"
  show False = "no"
instance Read Bool where
  read "yes" = True
  read "no"  = False
  read s     = error ("could not read as Bool: " ++ show s)
```

(Note '++' is the string concatenation operator in Haskell.) Assuming sensible implementations of Show and Read exist for the Int type, uses of read and show dispatch to the correct implementation upon their use.

```
> show True ++ " boss"
"yes boss"
> show 123 ++ show 456
"123456"
> 1 + read "10"
11
> False || read "yes"
True
> show 1
"1"
> read "1"
error: ambiguous type variable 'a' in the constraint 'Read a'
> read "1" :: Int
```

[1]It is recognized that object oriented dispatch may take place at compile time, and that type class dispatch may take place at runtime. These are merely typifications of mainstream usage.

[2]these are not equivalent to the instances defined in the Haskell Prelude.

```
1
> read "1" :: Bool
error: could not read as Bool: "1"
```

Because read has type (Read a) => String -> a — which may be read “for any type a which is an instance of Read, read has type (String -> a)” — the read function to be used at a given call site must be determined by type context. Context dependent functions like read may also be used and composed to create other context dependent functions.

```
showWithNewline :: (Show a) => a -> String
showWithNewline e = show e ++ "\n"
```

Type classes thus provide a way for the programmer to use a common interface of functionality where the burden of implementation lies on the *type*s to which it is projected. This mirrors object oriented programming, where a programmer may use a common interface of functionality where the burden of implementation lies on the *value*s to which it is projected.

1.1.2 Monads

Monads have lately appreciated a large mass of attention with regard to their usefulness in programming. As a purely abstract mathematical idea realized inside programming practice, a monad is any type or entity which provides the following interface:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

The >>= function is typically called “shove” or “bind” because it allows for some pure value *a* to be accessed, so long as its use results in another monadic value. The return function defines how any pure value may be encapsulated inside a monadic value which “does nothing” in the semantics of the monad. More specifically, return should obey the following laws:

```
          xM >>= \x -> return x ≅ xM
return () >>= \_ -> xM          ≅ xM
```

One of the simplest instance of the Monad class is the Maybe type.

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
  (Just a) >>= f = f a
  Nothing  >>= f = Nothing
  return x       = Just x
```

As a monad, the Maybe type provides a simple control flow mechanism that will terminate computation on the inspection of the Nothing value.

```
> Just 1 >>= (\x -> return (x + 2))
Just 3
> Nothing >>= (\x -> return (x + 2))
Nothing
> Just 1 >>= (\x -> Nothing) >>= (\y -> error "some error")
Nothing
```

Full coverage of monads will not be provided at length, however references to their "usefulness" will be somewhat taken for granted throughout this paper. This is due to the nature of their complexity and the perceived lack of resistance to the claim that "monads are useful" from those who use or understand them in depth.

1.1.3 Monad Transformers

Monad transformers are monads which are able to be composed with another monad in order to obtain the functionality of both. For example, a programmer may wish to use a Parser and State monad together to achieve a parser which has access to some arbitrary state as it parses. Because of the difficulties of "escaping" from a monad — or in the case of the IO monad where "escaping" is not possible — it is usually troublesome to try to "weave" the computation of two monads together. Monad transformers provide a standard way to compose monads, so that any monadic value inside a monad composition may be *lift*ed to the common monadic composition type.

At the type level, a monad transformer is any monad which provides the following interface:

```
class Monad t => MonadTransformer t where
  lift :: Monad m => m a -> t m a
```

This allows for any monad to be wrapped inside any other monadic type that instantiates MonadTransformer. A tower of monads can then be constructed where a monad of any depth can be *lift*ed to outer type.

Monad Transformers will also not be discussed at length and, as with monads, references to their "usefulness" will be somewhat taken for granted throughout this paper.

1.1.4 Why Monads and Monad Transformers?

Because monads, and to a greater extend monad transformers, are truly abstract interfaces, they rely heavily on type classes to overload the meaning of functions >>=, return, and lift . It is these clean abstractions for abstract computation which type classes enable that motivates the implementation of "something like type classes" in languages that do not feature a comparably powerful type system.

Despite their usefulness, monads and monad transformers have not been adopted in full in other languages because these languages lack a sufficient mechanism of semantic dispatch. Bringing the power of type classes to these languages is thus motivated by the resulting ability to express the monad and monad transformer interfaces in ways that are currently not possible.

1.2 Operator Overloading

In modern programming languages it is common for the meaning of a function or primitive operation to be *overloaded* to have different semantics depending on context. For example, in the sequence of C++ statements

```
MyObject x; MyObject y; MyObject result = x + y;
```

the + operator may be defined by the programmer as a method for the *MyObject* type, overloading the meaning of + to be something other than numeric addition. In languages that support object oriented programming, operator overloading is typically achieved in this way; by mapping operators to object methods. This approach to operator overloading falls short in at least the following areas:

1. Operators that wish to be overloaded do not by nature depend uniquely on the value of the first argument. The object oriented programming approach to operator overloading artificially requires this constraint.

2. *Values* that which are context dependent are not capable of being expressed. The + operator may be overloaded for both a set and list data type, but the value *empty* may not be overloaded, which would be useful as a generic identify for identity for +.

Type classes are the latest consensus on how to properly support operator overloading in a functional programming language. They play nicely with type inference and, among other important things, allow for the proper overloading of binary functions and the overloading of values based on context. The Haskell code

```
class Container a where
  (+++)   :: a -> a -> a
  empty   :: a
```

defines a Container type class that allows the overloading of +++ and *empty*,

```
instance Container [a] where
  (+++)  = (++)
  empty  = []
instance Container (Set a) where
  (+++)  = Set.union
  empty  = Set.empty
```

defines overloaded functionality of +++ and *empty* for lists and sets, and

```
let x = [1, 2, 3]              :: [Int]
    y = Set.fromList [7, 8, 9] :: Set Int
in (x +++ empty, y +++ empty) :: ([Int], Set Int)
```

demonstrates a valid use of the overloaded operators.

Of special importance is the ability to overload the value empty using type classes. The implementation of empty chosen in a given context is based on the resulting type of the expression. More importantly, it would *not* be possible for the implementation of empty to be dynamically selected based on a value. Consider a function containerFold which combines an arbitrary number of containers, making use of +++ and empty:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f i []     = i
foldl f i (a:as) = foldl f (f i a) as

containerFold :: (Container a) => [a] -> a
containerFold = foldl (+++) empty
```

We expect the following results when combining zero number of containers:

```
containerFold [] :: [Int]   = []
containerFold [] :: Set Int = Set.empty
```

In the above two expressions, the values [] and Set.empty are not able to be selected dynamically from a value, as no values of type [Int] or Set Int are present.

Functions like containerFold are not able to be expressed using traditional object oriented approaches to operator overloading because the selection of a function or value is not able to depend on some existing value. Type classes allow for functions like containerFold to be expressed because they do not depend on runtime values for the selection of functions. Rather, they depend on compile time types, which are always present, even when values are not.

1.3 Functions on Abstract Members of Type Classes

As seen previously, with type classes it is possible to define functions that are dependent on the existence of a particular type class context. Consider the simplified versions of the following type classes defined in Haskell

```
class Monoid m where
  mempty   :: m
  mappend  :: m -> m -> m
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

and instances

```
data All = All { getAll :: Bool }
data Tree a = Leaf | Node a (Tree a) (Tree a)
instance Monoid [a] where
  mempty  = []
  mappend = (++)
instance Monoid All where
  mempty  = All True
  mappend (All b1) (All b2) = All (b1 && b2)
instance Foldable [] where
  foldr f i [] = i
  foldr f i (x:xs) = f x (foldr f i xs)
instance Foldable Tree where
  foldr f i Leaf = i
  foldr f i (Node x left right) =
    foldr f (f x (foldr f i right)) left
```

Using type classes, we are able to write an even more generic fold than was shown in the previous section:

```
fold :: (Monoid m, Foldable t) => t m -> m
fold = foldr mappend mempty
```

which can be read "for any foldable type t and monoidal type m, we can define $fold$ which has type $t\ m -> m$". This demonstrates the power of abstraction that type classes provide. The $fold$ operator becomes overloaded based on a rather complex context: that we have a foldable data structure containing elements defined to be monoidal. Expressions like

```
getAll (fold (map All [True, True, True, False]))
```

and

```
fold (Node [1, 2] (Node [3, 4] Leaf Leaf)
                  (Node [5, 6] Leaf Leaf))
```

have the expected functionality and evaluate to False and [3, 4, 1, 2, 5, 6] respectively.

Type classes thus scale in a way that object oriented method selection does not. With type classes, dispatch may depend on an arbitrarily complex context, where with method selection, and therefore operator overloading that leverages method selection, dispatch must depend uniquely on a single value.

1.4 Monads and Monad Transformers

Few programming abstractions benefit from the existence of type classes as do the use of monads and monad transformers. When writing programs that need to pass around some state variable, syntactic support for monads in Haskell allows a programmer to write

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()

increment :: MonadState Int m => m ()
increment = do
  x <- get
  put (x + 1)
```

which increments an integer, which is the state being threaded through some arbitrary computation. Because *get* and *put* are overloaded type class functions, *increment* becomes a context dependent function defined for any monad that instances MonadState. The ability to express a stateful computation in terms of *get* and *put* abstractly is valuable in that monads can be combined using monad transformers to implement these interfaces in different ways. Consider another common monad interface, MonadWriter, which abstracts over the ability to output results that automatically get combined as a monoid.

```
class (Monad m, Monoid w) => MonadWriter m w | m -> w where
  tell :: w -> m ()
```

It is possible to write functions dependent on either one or both abstract monad contexts.

```
double :: MonadState Int m => m ()
double = do
  x <- get
  put (x * x)
```

```
report :: MonadWriter String m => Int -> m ()
report x =
  tell ("reporting " ++ show x)

doubleAndReport :: (MonadState Int m, MonadWriter String m) => m ()
doubleAndReport = do
  double
  x <- get
  report x
```

Monad Transformers allow for the combining of State and Writer monads that support both State and Writer type classes, allowing *doubleAndReport* to be used as a combined monadic value. Type classes and type inference allow for programs to be written in these extremely abstract ways, where actual implementations of *get*, *put*, and *tell* are chosen differently by context. Notice that *doubleAndReport* is not a function, but rather a single monadic value. The actual implementation of the value *doubleAndReport* will vary widely between each choice and use of monad. This is therefore another example of dispatch which *cannot* uniquely depend on a value, and therefore requires a mechanism for dispatch that does not use values as arbiters of selection.

2 Aspect Classes

2.1 Basic Semantics and Usage

While it is now clear that if we are to express abstract operations more purely we must remove the connection between dispatch and runtime values, it is not clear how such dispatch could be achieved without a type system. *Aspect classes* come out of the realization that the meaning of a function can be dependent on a context which is not determined by the examination of types. This removes the need for a complex type system in order for aspect classes to exist, and greatly simplifies the semantic weight of their use. *Aspects* are then the things which become instances of a given aspect class, the way a type may become an instance of a type class. Without the conceptual underpinnings of a *types*, aspects become a more general representation of some *concrete* idea, and aspect classes a more general representation of some *abstract* idea.

Take, for instance, the abstract concept of a $field$ in mathematics. Besides a set of well defined axioms, a $field$ is a truly abstract concept, that has a number of well known concrete instances. One such concrete instance, the set of rational numbers, could be then be described as an *aspect* that is an instance of the *field aspect class*. Another simple aspect class is *show*, representing the abstract concept that something has a simple textual description. The set of rational numbers can then also be an instance of the *show* aspect class, because there is a simple textual representation for any given rational number.

Abstract concepts, and therefore *aspect classes*, often build on one another. For instance a *field* is also always a *ring* in mathematics, so any aspect that is an instance of *field* must be forced to also be an instance of *ring*. Conceptually, when talking about "the field of rational numbers", uses of the rational numbers as a ring is implied to be consistent. Type class declarations allow type classes to extend any number of other type classes, and this necessity holds when types are removed. Aspect classes therefore support this conceptual extension of one concept to form another.

Concrete concepts, and therefore *aspects*, often depend on the existence of other aspect class instances. For instance allowing the aspect *list* to be an instance of the *show* aspect class might depend on the instance of *show* for some aspect that represents the values contained in the list. While this looks suspiciously like type classes, where displaying an element of type list depends on the ability to display the type of element it encloses, aspect classes place no correlation between the types of things and the functions that are supplied by a given context.

To see how aspect classes may be used more directly, let us first look at some simple uses of aspect classes.

```
(define-aspect number<>)
(define-aspect rational-number<>)
(define-aspect list<>)

(define-aspect-class (show~ a)
 (define show))

(define-aspect-instance (show~ number<>)
 (define (show n)
  (format "~a" n)))
(define-aspect-instance (show~ rational-number<>)
 (define (show n)
  (format "~a/~a" (inexact->exact (numerator n))
                  (inexact->exact (denominator n)))))
(define-aspect-instance (show~ (list<> a)) requires ([show~ a])
 (define (show xs)
  (if (empty? xs)
        "[]"
        (let ([tail (show (cdr xs))])
         (using-context (show~ a)
          (format "~a:~a" (show (car xs)) tail))))))

(using-context (show~ (list<> number<>))
 (show '(1.5 2.5 3.5))) ; => "1.5:2.5:3.5:[]"
(using-context (show~ (list<> rational-number<>))
 (show '(1.5 2.5 3.5))) ; => "3/2:5/2:7/2:[]"
```

A single aspect class show is defined, with instances defined for number<>, rational−number<>, and list <>. The instance definition for showing a list requires that the aspect it is applied to is also an instance of the show aspect class. When specifying a context to use, list <> must be applied to some other aspect in order to instantiate its instance for show. With the use of aspect classes that extend on another, and aspect instances that depend on other instances, a very abstract universe can be constructed and linked against for any problem domain. Also of note is the fact that in the above example both uses of *show* are applied to the same value (and therefore same *type* of value), but yield different resulting values. This is a capability not possessed by both value based dispatch and type based dispatch methods: that the overloading of a function can be independent of both value and type context. Rather, dispatch is more explicitly selected from the "aspect context" in which the programmer places an expression.

2.2 Comparison with Type Classes

Aspect classes differ from type classes in that they do not by necessity correspond to the type of a particular value. Because of this, and the lack of interaction with type inference, there are no restrictions on how a type of data may interface with an aspect class. Consider, for instance, the type of a state monad: State s a ::= (s −> (a, s)). Because not every function of type (a −> b) is monadic, a newtype wrapper must be introduced in order to specify that only functions of the form (s −> (a, s)) are to be called state monads.

```
newtype State s a = State (s -> (a, s))
instance Monad (State s) where ...
```

With aspect classes, the universal nature of the function type is not problematic. Defining the state monad using aspect classes only requires the introduction of the aspect state−monad<> and the appropriate definitions for >>= and return, which are free to work directly with unwrapped functions.

Aspect classes do not interact with a type system. The negative implication of this is that type inference can not be used to infer the context that should be used to instantiate a class field. However, advanced uses of type classes often require the programmer to explicitly supply type signatures in order to resolve an ambiguous type context. Uses of aspect classes to explicitly specify a context are similar to that of type classes in this respect.

2.3 Aspect Families

In the context of type classes, type families allow for relationships between types that can be referenced in the type signatures of class declarations and context dependent functions. For instance

```
type family Scalar a :: *
class NormalVectorSpace s where
  (<+>) :: s -> s -> s
  (<->) :: s -> s -> s
  (<*)  :: s -> Scalar s -> s
  norm  :: s -> Scalar s

normalize :: (NormalVectorSpace s, Fractional n,
              Scalar s ~ n)
           => s -> Scalar s
normalize v = v <* (1 / norm v)
```

defines a type class NormalVectorSpace which uses the type family Scalar in a way that resembles type level function application[3]. The type signature of normalize may be read "for some type *s*, such that *s* is defined to be a *NormalVectorSpace*, and for some type *n*, such that *n* is defined to be a *Fractional*, and that the type *Scalar s* is equivalent at the type level to *n*, *normalize* is of type s −> Scalar s" Because the use of aspect classes evades the need to declare types, the idea of aspect families initially appear less useful or necessary than that of type families. However, function-like relationships between aspects can be useful for revealing unexposed aspects in an evaluated aspect application. In an application similar to the Haskell demonstration above, aspect families can aide in defining functions that require the context of seemingly unrelated aspects.

```
(define-aspect-family scalar<*>)
(define-aspect-class (normal-vector-space~ a)
```

[3]as apposed to type construction

```
 (define <+>)
 (define <->)
 (define <*)
 (define norm))

(define-contextual (show-norm v) (s) requires (normal-vector-space~ s)
 (using-context (show~ (scalar<*> s))
  (show (<* v (/ 1 (norm v))))))
```

3 Aspect Classes In PLT Scheme

3.1 Syntactic Forms Introduced by Aspect Classes

Aspect classes are implemented in PLT Scheme as primitive[4] syntactic forms through the use of macros that share information across modules. As for any syntactic extension to a language, it is important that great effort is spent in an attempt to find the right set of primitives. As described later, the seemingly complex syntactic form define−contextual is expressible in terms of define−aspect define−aspect−class and define−aspect−instance. The primitives introduced by aspect classes are define−aspect, define−aspect−alias, define−aspect−class, define−aspect−instance, using−context, and reveal−context.

Primitives *define-aspect* and *define-aspect-alias* are introduced for defining and aliasing aspects.

```
(define-aspect name)
(define-aspect-alias name aspect-expr)
```

In the compile time environment, aspects contain data mapping *context-value-path*s to both *context-instance-template*s and *resolved-instance-identifier*s, where

$$\begin{aligned}
\textit{aspect} &::= \textit{symbol}\\
\textit{aspect-class} &::= \textit{symbol}\\
\textit{aspect-family} &::= \textit{symbol}\\
\textit{aspect-value} &::= \textit{aspect} \mid (\textit{aspect aspect-value} \ldots)\\
\textit{context-value} &::= (\textit{aspect-class aspect-value} \ldots)\\
\textit{aspect-expression} &::= \textit{aspect-value}\\
&\mid\ (\textit{aspect-family aspect-value} \ldots)\\
\textit{context-value-path} &::= (\textit{aspect-class aspect} \ldots)
\end{aligned}$$

[4]primitive in the sense that they do not directly desugar to other ‘primitive’ forms

*Context-instance-template*s contain templates of function definitions and will be compiled against different contexts as needed. *Resolved-instance-identifier*s reference the compiled version of a template for a specific context. Symbols instead of syntax objects are used, meaning lexical scope for aspect identifiers is not tracked and that there is only one top level namespace for aspects and classes across modules[5]. However, aspect aliases can be used to give lexical bindings of aspect values.

Primitive *define-aspect-class* is introduced for defining aspect classes.

```
(define-aspect-class context-pattern requires (context-expression ...)
 class-field-defn ...)
```

Aspect classes store their defined *context-pattern*, dependent *context-expression*s, a list of fields that an instance of that class must implement, and default instance templates where provided, where

$$\begin{aligned}
\textit{context-pattern} &::= \textit{aspect-class} \mid (\textit{aspect-class var} \ldots) \\
\textit{context-expression} &::= (\textit{aspect-class aspect-expr} \,..) \\
\textit{class-field-defn} &::= (\text{define } \textit{field} \; [\textit{default-value}]) \\
\textit{var} &::= \textit{symbol not otherwise mentioned}
\end{aligned}$$

Primitive *define-aspect-instance* is introduced for defining instances of aspect classes.

```
(define-aspect-instance (aspect-class aspect-pattern ...)
   requires (context-expression ...)
 (define field value) ...)
```

Defining an aspect instance mutates the mappings from *context-value-path*s to *context-instance-template*s to associate the specified *instance-path* with a template for field definitions in the first aspect mentioned in an aspect-pattern, where

$$\textit{aspect-pattern} ::= \textit{aspect} \mid (\textit{aspect var} \ldots)$$

Primitive *using-context* is introduced for the construction and use of resolved instances.

```
(using-context context-expression expr ...)
```

[5]using syntax objects to identify aspects and aspect classes would require the keys of these maps to be lists of syntax objects, which greatly complicates the lookup of a context-value-path

All fields associated with the aspect class and its class dependencies will be bound to concrete implementations[6] to be used by expressions within the using-context scope.

Primitive *reveal-context* is introduced to allow for the exposing of an enclosing context to a lexically scoped binding.

```
(reveal-context context-pattern expr ...)
```

Pattern variables mentioned in the context-pattern will be valid identifier bindings for use in aspect-expressions within the reveal−context form.

Syntactic form *define-contextual* is introduced to allow the defining of context-dependent values.

```
(define-contextual name (var ...) requires (context-expression ...)
 value)
```

Defining a contextual value can be achieved by creating a new aspect and a new aspect class of one field of which the newly generated aspect is an instance. The field is then bound to a syntax transformer that, when expanded, discovers the current context for each instance dependency using reveal−context, and evaluates to the concrete implementation of the value for that context.

```
(define-syntax define-contextual
 (syntax-rules
  [(_ name (var ...) requires (context-expression ...)
    value)
   (begin
    (define-aspect custom-aspect)
    (define-aspect-class (custom-class a)
     (define custom-name))
    (define-aspect-instance (custom-class (custom-aspect var ...))
       requires (context-expression ...)
     (define custom-name value))
    (define-syntax name
     (... expand to ...
      (reveal-context* (context-expression ...)
       (using-context (custom-class (custom-aspect var ...)))
        custom-name))))]))
```

[6]as we will see in the next section the compilation of these concrete forms are delayed until their actual use

3.2 Performance Implications

In order to achieve sensible compile times of programs using aspect classes, it is essential the compilation of aspect class instances is delayed and that the resulting concrete implementation is cached.

To see why delaying compilation is necessary we first recognize the fact that it is grossly impractical to pre-compile the implementations of all possible combinations of aspects that may instance any given aspect class. For instance, in an environment where there are many aspects which are instances of a show aspect class, and many container-like aspects for which their instance of show depends on the instance of show for what it contains, it would be unwise to pre-compile all possible concrete instances of show. This is because in general a program will use an extremely small subset of possible instance combinations. In fact, there are actually an infinite number of possible contexts, seeing as how they may be used in a self recursive way. For example, the instance for the aspect class show˜ of the aspect list<> defined in section 2 makes use of the instance of show˜ of its applied context variable. An expression may contain an arbitrarily deep application tree of list<> aspects, as in:

```
(using−context (show˜ (list<> (list<> (list<> number<>))))
 (show '(((4)))))
```

It is also unwise to pre-compile concrete instances at the site of a using−context form. This is because it is common for the expressions in a using−context form to underuse all the fields brought into scope by the dependencies of the context. In a common case where the monad˜ aspect class depends on the applicative˜ aspect class, which depends on the functor˜ aspect class, not all fields introduced by the hierarchy of classes need be compiled as typically not all will be used. For example the expression:

```
(using−context (monad˜ my−monad<>)
 (return #t))
```

should not force the compilation of the applicative˜ and functor˜ instances of my−monad<>. The motivation to not compile all concrete instances on the site of a using−context form is further enhanced upon the realization that, in general, the depth of inheritance of an aspect class should not directly increase the amount of compile time work.

Caching the compiled forms of concrete instances is necessary for the more obvious reason that multiple uses of a context should be able to share a single compiled version of their implementation. Using this approach, a using−context form merely parameterizes an association between an aspect class and aspect expression for the enclosed expressions. The use of a field of an aspect class will look up the aspect expression associated with its aspect class and force the compilation of its implementation if it has not yet been done. The actual realization of contexts are delayed in this way until the actual use of a field, resulting in no unnecessary compilation.

4 Related Work

An attempt in bringing type classes to scheme has previously been formulated by Garcia and Lumsdaine[reference]. Their approach satisfied simple operator overloading in scheme using value based dispatch with predicate functions and a lookup table. Expressing dependencies or relationships between instances of type classes is not possible in their implementation. As a result, no foundation is provided for the use of monads, monad transformers, or other comparable forms of abstract function definition.

Aspect classes could conceivable be extended to support the specification of context dependencies at the module level, resembling something very similar to Units as introduced by Flatt[reference].

5 Future Work

Currently both type classes and aspect classes require a context to be determined at compile time, forfeiting the advantages that dynamically resolving a context at run time could provide. Because type classes have never been used in such a dynamic way, it is difficult to immediately discuss the usefulness of a type class language feature which interacts with runtime values. Making aspect classes first class values which may be manipulated and dispatched upon at runtime would free them from their reliance on being wholly described in a compile time environment. It is unclear how complex the interaction between run time and compile time aspects could become. However, the exploration of run time aspect classes, specifically their common use cases, could raise more discussion on what exactly the dynamic dispatch of a function or value should depend on to achieve the most flexibility and power in expressing abstract computation.

It is possible that aspect classes could exist in a language where type classes already exist in conjunction with type inference. A correlation between an aspect and a type could be declared for some aspects, allowing for the context of an aspect class to be inferred. This would allow for aspects which do not depend on a type to exist in harmony with those that do, simultaneously reaping the benefits of inferred contexts and type independent dispatch.

6 Conclusion

Type classes are an invaluable tool in the hands of a programmer who wishes to describe computation in the abstract. This paper introduces aspect classes as an alternative for the abstract definition of values which does not depend on type inference. With aspect classes, it is now possible for users of dynamically typed languages like Scheme to enjoy in full the usefulness of programming abstractions like monads and monad transformers.

It is possible that type classes as they exist in Haskell are a restricted use of aspect classes, where every aspect corresponds to an inferable type. Wether or not this is the case, incorporating aspect classes back into Haskell is an example of why having an extensible macro system matters. If type classes are somehow a unification between a type and an aspect class, then they should be generalized as such. The ability to find a better primitive for a core feature of a language should not be held back by the inability to extend the syntax of that language, building syntactic forms upon one another.