Conditional Dispatch with Type Classes

Jean Yang
MIT CSAIL
jeanyang@csail.mit.edu

Paul Govereau Greg Morrisett
Harvard University
{govereau, greg}@eecs.harvard.edu

Abstract

We propose a *conditional dispatch* operator for constrained polymorphic definitions which applies function f to argument v if and only if v's type is in a class that supports f, and returns a default value otherwise. Such a mechanism is particularly useful in Haskell, where type classes provide the basis for ad hoc polymorphism. In this paper we describe our **dispatch** operator and its implementation, which we support with an extension to Template Haskell and our **ClassDynamic** library. We show that by extending Template Haskell to allow greater access to the compile-time type-checking environment, one can easily implement such constructs in libraries rather than as language extensions. Our work has two main implications: 1) it is useful to have type-class-based reflection and 2) extending Template Haskell to allow more access to the compile-time environment allows for useful features to be implemented as libraries rather than compiler extensions.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Type analysis is useful for handling structure-specific details of generic programming and for effective programming with dynamic types. *Type classes* are useful for defining classes of types that support a given set of behaviors and for allowing assumptions about the operations supported by types in polymorphic functions (Wadler and Blott 1989). We claim that type analysis *with* type classes allows for greater flexibility in dispatch than is provided by type-classes alone, as well as better support for programming with dynamic types.

Suppose we wish to implement a set over some abstract element type. At a minimum, we need some equality operation for elements to implement the **member** function which determines membership in a set. Hence, it is natural to require the element type to be in the **Eq** class. However, if the element also provides an ordering operation, then we can implement sets more efficiently (e.g., using a balanced binary tree as opposed to a list.) Thus, for efficiency's sake, we might hope that elements satisfy the **Ord** class.

Ideally, we should be able to choose a set implementation based on what type-classes the element satisfies. Given such a facility for testing type-class membership, we might write the following **insert** function:

```
insert :: (Eq a) => a -> Set a -> Set a
insert x t_or_l =
  case cast (x :: (Ord a) => a) of
    Just x_ord -> insert_tree x_ord t_or_l
    Nothing    -> insert_list x l_or_l
```

This function tests whether the element type satisfies the **Ord** class and if so, uses the efficient tree-based representation. Otherwise, the function falls back on a simple list representation which only requires the **Eq** class.

Unfortunately, we cannot currently write such an implementation because we do not have a notion of type-class case in Haskell. We must instead decide whether to sacrifice the possibility of an efficient implementation or whether to bound the polymorphism of **insert** to the **Ord** class, which would require defining other ordering operators such as **max** and **min**, over any data type for which we would like to use **Set**. When faced with such a tradeoff, the reasonable library implementor is likely to choose the latter: in the Haskell **Set** library, the restriction is **Ord**.

In this paper, we propose a **dispatch** operation, which supports type-class conditional dispatch, and thereby addresses this issue. Our implementation examines the type class instances that have been defined for a program and encodes the possible dispatches using a universal representation similar to Haskell's type **Dynamic** . Our solution uses an extended version of Template Haskell for examining the static type-checker environment and a new library **ClassDynamic** to support conditional dispatch. In this paper we describe

1. the semantics of the **dispatch** operation,
2. an implementation of **dispatch** involving an extension to the Template Haskell processor of GHC and the **ClassDynamic** library, and
3. examples where **dispatch** is useful.

In Section 2, we go into more detail with the **Set** example and show our solution for it. We then describe examples and provide details on the implementation and the **ClassDynamic** library.

2. Conditional dispatch for the set example

In this section we describe the **Set** example in greater detail and show how to use conditional dispatch to achieve the desired goal of being able to use both **Eq** and **Ord** methods.

2.1 The type class restriction tradeoff

Consider the following interface for sets:

```
empty :: Set a
```

```
insert :: Eq a => a -> Set a -> Set a
member :: Eq a => a -> Set a
isSubsetOf :: Eq a => Set a => Set a => Bool
```

The context restriction **Eq** a requires that any argument of type a has (==) defined.

The context restriction also imposes an implementation restriction: the **Eq** a context restriction *only* allows dispatch of (==) on the arguments. To demonstrate this, let us consider implementing our set as a balanced tree. We can implement the **Set** data type to allow for either a tree or list representation:

```
data Set a = TreeRep (Tree a)
           | ListRep [a]
```

We can now have two different insertion routines, one for elements that have total ordering defined:

```
insert_tree :: (Ord a) => a -> Set a -> Set a
insert_tree x t_or_l =
  case t_or_l of
    TreeRep t -> TreeRep (Tree.insert x t)
    ListRep _ -> raise Impossible
```

which uses insertion routine shown in Figure 2, and one for elements that only have equality defined:

```
insert_list :: (Eq a) => a -> Set a -> Set a
insert_list x' l_or_l =
  case t_or_l of
    TreeRep _ -> raise Impossible
    ListRep l ->
      case l of
        [] -> [x']
        _:_ -> if elem x' l
                 then l
                 else x':l
```

To implement our set as a balanced tree, we would need a total ordering on the elements, which we would want the programmer to define because we cannot easily derive such an ordering in a programmatic way. Since we may only assume that the arguments have (==) defined with **Eq**, we must turn to **Ord**, which has methods (<) and (>). Unfortunately, we have no way of showing that we have an argument whose type is an **Ord** instance.

This reason why we cannot dispatch on **Ord** methods with an **Eq** restriction is that in order to show the compiler that we can dispatch a method of a type class C on a value v, we need to show that v has a type that has been defined as an instance of C. Currently, we have no way of expressing that x is an instance of the **Ord** class without performing type analysis and a cast to a specific type. This would require using Haskell's dynamic **cast** function, provided by the Data.**Typeable** library, and would look something like this:

```
insert x t_or_l =
  case (cast x :: Maybe Bool) of
    Just b -> insert_tree b t_or_l
    Nothing ->
      case (cast x :: Maybe Int) of
        Just i -> insert_tree i t_or_l
        Nothing -> ...
        -- And it goes on...
        -- For all instances...
          case (cast x :: Maybe String) of
            Just s -> insert_tree s t_or_l
            Nothing -> insert_list x t_or_l
```

Alas, this definitions is *closed* and cannot support further instances of the **Ord** class.

Because we must declare the type class context of the arguments in the function signature, we must make a choice between the sufficient context restriction and one desired for efficient implementation. In order to dispatch on **Ord** methods in the body of the function, we can either strengthen the context restriction from **Eq** to **Ord** or perform type analysis in the body of the function. Since the type analysis is necessarily closed to new type definitions and type classes are useful *because* they are open to new type definitions, the latter method will not do.

```
class Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Figure 1. Haskell library definition for **Ord** class.

```
insert :: Ord a => a -> Set a -> Set a
insert x t
  = case t of
      Tip -> singleton x
      Bin sz y l r
          -> case compare x y of
               LT -> balance y (insert x l) r
               GT -> balance y l (insert x r)
               EQ -> Bin sz x l r
```

Figure 2. Haskell **Set** library's **insert** function.

To use the Haskell **Set** library for her new data type, the programmer must now associate it with the **Ord** class which has many methods that are irrelevant to the fact that the value will be used in a set. This not only requires more work on the programmer's part, but violates the principle of modularity: details relevant only to the implementation are revealed in the interface.

2.2 Solution with conditional dispatch

Our class-based dispatch operators allow us to write the following **Set** implementation:

```
insert :: (Eq a) => a -> Set a -> Set a
insert x t_or_l =
  let fns = $(inst 'insert_tree)
      x' = toClassDyn x
      tl' = toClassDyn t_or_l
 in case fromClassDynamic
           (dispatch (dispatch fns x') tl') of
       Just v -> v
      Nothing -> insert_list x t_or_l
```

In the code above, the call $(**inst** ' insert_tree) is a Template Haskell call which builds a collection of specialized versions of the insert_tree function and returns the collection as a **ClassDynamic**. The **ClassDynamic** is represented as a list of **Dynamic** values (i.e., values paired with a representation of their type.) The versions are built by taking the set of all types that satisfy the **Ord** class, and by specializing insert_tree to those types. The **dispatch** operation takes two **ClassDynamic** values and attempts to calculate the cross product by applying every function in the first collection to every value in the second collection. The result is returned as a new, possibly empty **ClassDynamic**. In the example above, if the type a ends up being an instance of the **Ord** class, then fns will be a singleton set of the insert_tree operation specialized with a's **Ord** methods. Consequently, the first calls to **dispatch** on the singleton collections x' and tl' will result in a singleton value which we project using the **fromClassDynamic** function. On the other hand, if a does not implement the **Ord** interface, then fns will be the empty collection. In turn, this will mean that the dispatches return an empty collec-

tion, and the call to **fromClassDynamic** will yield **Nothing**. Thus, we invoke the default operation of inserting the value into a list.

3. Overview of conditional dispatch

We describe the semantics of the conditional dispatch operator **dispatch**. We implement codedispatch by examining the relevant type class instances associated with a bounded polymorphic function in order to determine whether a specific dispatch is valid. We also present the **ClassDynamic** library to support the type analysis required for resolving dispatches.

We use an extended version of Template Haskell to write a function that allows us to get all possible instantiations of a function signature:

```
inst :: Name -> Q Exp
```

The function **inst** takes a function and returns an expression that, when reified, is a value of type **ClassDynamic** which contains a representation of the instantiations of the function. Type Exp is a Template Haskell data type that reifies into expressions; Q is the Template Haskell monad which supports reification.

Consider the following type class with the following instances:

```
class SomeClass a where
  is_member :: a -> Bool

instance SomeClass Int where
  is_member _ = True
instance SomeClass Bool where
  is_member _ = False
```

The call $(**inst** 'is_member) would return the following value of type **ClassDynamic**:

```
combineClassDynamic
  [ toClassDyn (is_member :: Int -> Bool)
  , toClassDyn (is_member :: Bool -> Bool) ].
```

Type **ClassDynamic** stores a collection of **Dynamic** values; the codeClassDynamic library provides support for handling multiple representations of a value's associated type.

The **inst** function and the data type **ClassDynamic** allow us to write the following function:

```
dispatch :: ClassDynamic -> ClassDynamic
         -> ClassDynamic
```

This function takes a list of possible dispatches and applies all of the relevant ones to the argument. Note that the result could potentially contain an empty set of type representations.

We provide the details of Template Haskell and the implementation in Section 5.

3.1 Disambiguating the dispatch

Disambiguating the dispatch involves 1) determining whether we have a value whose type has an appropriate dispatch and 2) determining *which* function to dispatch. We can resolve both of these issues by examining the set of instances that have been defined and the possible instantiations of a function. The issues of conditional dispatch occur only when we have type class constraints: when there is a type class constraint, we know a function is not fully polymorphic but bounded to only a set of types. Given a value, we must determine if we have a valid type in the set, and which dispatch should occur.

Given the declared instances of a function, we know all of the possible instantiations, or ways to dispatch on, a given function signature. If we know the type of a value, we can then examine this set to determine the dispatch.

3.2 Building on Dynamic

We a new type **ClassDynamic**, which we can think of as representing a list of **Dynamic** values, to store a collection of possible values because we need to represent the multiple possible representations when the dispatch is ambiguous. There is ambiguity whenever a function's there are type variables that do not appear in a function's first argument. This is the case with both of the following functions:

```
some_function :: (SomeClass a b) => a -> b -> Int
show_with_level :: (Show a) => Int -> a -> String
```

Once a function is fully applied the dispatch is necessarily unambiguous, so if we have some way of representing ambiguity in partially applied functions the final result will be the correct one. We describe the **ClassDynamic** library more in Section 6.

3.3 Solution in Haskell framework

To get the appropriate instances of a function, we take advantage of Template Haskell's capabilities for reifying abstract syntax and accessing the compiler environment (Sheard and Peyton Jones 2002). In Section 5 we describe our extension to the Glasgow Haskell Compiler to gain sufficient information from the type checker environment.

To store such a dictionary and to perform type analysis, we need to 1) have a type representation that allows us to have values with types that are uncertain and 2) be able to perform type analysis. We use Haskell's **Typeable** library, which allows us to access type representations of values at run time with the **Typeable** class (Baars and Swierstra 2002):

```
class Typeable a where
  typeOf :: a -> TypeRep
```

TypeRep is a data constructor containing a runtime representation of an object's type. Haskell's **Typeable** library gives us the type-safe cast function:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b.
```

This function allows us to examine the type representations of `a` and `b` and return the result of the cast *iff* `a` and `b` are structurally equivalent. The type information from `b` comes from the static type-checking environment. Haskell's type **Dynamic** uses **Typeable** to store values with their type representations in order to express types whose values may not be known until run time. The library has functions that allow the user to convert to and from a **Dynamic** value, get the type representation of a **Dynamic**, and apply **Dynamic** functions to **Dynamic** values.

4. Other uses for class-based method dispatch

We describe the use of conditional dispatch for heterogenous list, for safe dynamic dispatch, and for conditional use of type class methods depending on context.

4.1 First class type qualifiers for heterogeneous lists

Kiselyov et. al. describe the shortcoming of Haskell support for heterogeneous lists, citing using lists of type [**Dynamic**] as an unsatisfying solution (Kiselyov et al. 2004). We present a way of handling heterogeneous lists that provides more information about list elements than using type **Dynamic**.

Consider a design pattern that involves taking heterogeneous elements associated with some operation and performing the operation on them. For instance, suppose we were making a graphical user interface and had different kinds of widgets, each associated with their own functions for displaying. We could define a type class for capturing this information:

```
class Display a where
  display :: a -> DisplayInfo
```

where DisplayInfo is a data type that contains information for a rendering procedure.

If we had a variable number of objects we might want to display, it is reasonable to have a diplay function take a list of such widgets. In Haskell we are allowed to write the following type signature:

```
display_all :: (Display a) => [a] -> ()
```

Unfortunately, this type requires that all items in the list have the *same* type. The type signature we want is

```
display_all :: [(Display a) => a] -> ()
```

Haskell does not support this because it only allows qualifiers to appear at the head of the expression.

According to the theory behind System F, we should be able to write our desired type signature. In System F $_\omega$ with qualified types, we have types of the following form (Jones 1992):

$$\sigma ::= \tau \mid \sigma \to \sigma \mid \forall\tau.\sigma \mid \pi \Rightarrow \sigma$$

According to this definition, we should be able to have *impredicative qualified types*. By impredicative, we mean that for a type such as

$$\tau = \forall\alpha.\alpha \to \alpha,$$

α is allowed to range over all types, including τ (Pierce 2002). Because a qualified type is a legitimate type, which should be able to write the type

$$[\pi \Rightarrow \alpha],$$

a list where each type α satisfies the type qualifier π.

We can use class-based method dispatch to simulate having such lists. We can have a function

```
display_all :: [ClassDynamic] -> ()
display_all elts = map try_display elts >> ()
where
  try_display :: ClassDynamic -> ()
  try_display e =
    case fromClassDynamic (dispatch $(inst '
        display) e) of
      Just a  -> a
      Nothing -> ()
```

This function gives us a way of displaying only the displayable elements in the list without having to do specific type-cases.

4.2 Safe dynamic dispatch

There is suboptimal support for dynamic dispatch in Haskell, as it requires explicit type-casting to monomorphic types, a process that is error-prone and full of boilerplate. Conditional function dispatch allows us to perform safe dynamic dispatch by dispatching on types based on membership in the appropriate classes.

We illustrate the convenience of safe static dispatch and the need for safe dynamic dispatch with an anecdotal example from Abelson and Sussman's *Structure and Interpretation of Computer Programs* (Abelson and Sussman 1996).

The premise of the exercise is that Insatiable Enterprises, Inc., "a highly decentralized" company with "a large number of independent divisions," has just been interconnected in a way that allows any user to regard the entire network as a single computer. The problem is this:

> Insatiable's president, in her first attempt to exploit the ability of the network to extract administrative information from division files, is dismayed to discover that...the particular data structure used varies from division to division. A meeting of division managers is hastily called to search for a strategy to integrate the files that will satisfy headquarters' needs while preserving the existing autonomy of the divisions.

The exercise says to assume each division has its own way of keeping personnel files, each personnel file has its own way of keeping records, and each record has its own structure. The reader's assignment is to implement get−record and get−salary procedures for the employee records of each division and a **find**−employee−record procedure that finds the record for a given employee. The exercise also asks the reader to think about how the central system should incorporate changes when Insatiable takes over a new company.

To solve this problem in Haskell, we could define the following type classes:

```
class Record r where
  getSalary  :: r -> Int

class PersonnelFile (Record r) => f r where
  getRecord :: f -> r

class FileCabinet (Record r) => c r where
  findRecord :: c -> EmployeeID -> Maybe r
```

We have declared a class Record over types that can tell you about the salary the record stores, a class PersonnelFile over types that can give you the record it stores, and a class FileCabinet over types that can eventually produce a record given some value of type EmployeeID. (We assume that EmployeeID is a predefined type.)

For the example implementation below we would declare the following instances:

```
data DivRecord = DivRecord { salary :: Int }
data DivFile   = DivFile   { employee :: Int
                           , record   ::
                               DivRecord }
newtype Cabinet0 = Map EmployeeID DivFile

instance Record DivRecord where
  getSalary = salary
instance PersonnelFile DivFile where
  getRecord = record
instance FileCabinet Cabinet0 DivRecord where
  findRecord fc eid =
    case lookup eid fc of
      Just file -> getRecord file
      Nothing   -> Nothing
```

With this solution, we will only be allowed to call the type class methods on types that are statically guaranteed to support the appropriate dispatches. The catch is that we are on the network, and it is not clear how we may be getting this information. Since the network by nature uses a uniform represenation for values, in using Haskell we will need a way to convince the static type checker that we will be using a value of the appropriate type.

To support types that are not known until run time we can use Haskell's **Typeable** type class, which allows intensional type analysis. For instances in the class of type α, **Typeable** requires the definition of the function **typeOf** :: $\alpha \to$ TypeRep. The presence of the type representation allows for the definition of type **Dynamic**, a universal type with a constructor of type $\alpha \to$ **Dynamic** that allows marshalling in and out of a representation of a value with its type representation.

To get the representation of a data type instance of the FileCabinet , we can write a function that looks something like the following:

```
fileCabinetFromDyn :: (Record r)
                   => Dynamic -> EmployeeID
                   -> Maybe r
fileCabinetFromDyn v eid = case fromDynamic v of
```

```
  Just v' :: Cabinet0 -> Just v'
  Nothing ->
    case fromDynamic v of
      Just v' :: Cabinet1 -> getRecord v' eid
      Nothing -> ...
                   Nothing -> Nothing
```

This solution is undesirable because 1) it involves a good deal of boilerplate, 2) the solution is closed–each new type class definition involves an additional case here, and 3) it is prone to programmer error, as it relies on the programmer to correctly enumerate all types it expects here. Though Haskell provides nice support for defining classes of types for purposes of determining dispatch, we can get none of these benefits when programming with type **Dynamic**. Dynamic dispatch in Haskell currently involves either knowing the exact type of the dynamic value in question or producing boilerplate code to determine the type.

Our **ClassDynamic** library has a function typeclasscase that would allow the definition

```
fileCabinetFromDyn' :: (Record r)
                    => ClassDynamic -> EmployeeID
                    -> Maybe r
fileCabinetFromDyn' v eid =
  fromClassDynamic
    (dispatch
      (dispatch
        $(inst 'findRecord) (toClassDyn v))
      (toClassDynamic eid))
```

The **dispatch** operator is a function defined in terms of Template Haskell functionality that looks up the instances of the FileCabinet class at in the compile time type environment and constructs the corresponding conversions. This is a better model in a large, decentralized system such as Insatiable's because it leaves the definitions of these cases open.

4.3 Conditional use of type class methods

Conditional dispatch is useful for when we want to use the defined functions for all type class instances and define some other default behavior.

On the Haskell Wiki, Keslyov and Peyton Jones address the issue of declaring a type-class instance based on context (Keslyov and Jones April 2008). The problem they address is that of having a default operation for types that are not defined as part of the class. They give the following example for trying to reuse **Show** functions for Print as illegal in Haskell code. This is wrong because the heads of the two instances declarations are the same:

```
instance Show a => Print a where
  print x = putStrLn (show x)
instance           Print a where
  print x = putStrLn "No show method"
```

They provide a solution using functional dependencies and overlapping instances, using an auxillary class Print' and a new class ShowPred that has no methods but behaves the same as **Show**. This solution involves declaring instances for ShowPred for the possible flag types, HTrue and HFalse, and then writing the following non-overlapping instances for Print':

```
instance (Show a) => Print' HTrue a where
  print' _ x = putStrLn (show x)
instance Print' HFalse a where
  print' _ x = putStrLn "No show method"
```

Our solution allows you to write the following:

```
print :: (Typeable a) => a -> IO ()
print x =
  case fromDynamic
         (dispatch $(inst 'show)
                   (toClassDyn x)) of
    Just str -> putStrLn str
    Nothing  -> putStrLn "No show method"
```

5. Implementation

We have implemented

1. a patch to the Glasgow Haskell Compiler (GHC) that provides Template Haskell with the possible instantiations of a function given its name, and
2. the **ClassDynamic** library, which has an implementation relying on the patch to Template Haskell.

We have been working with GHC 6.11 and our extension of Template Haskell's compile-time meta-programming capabilities.

We had an initial implementation involving a source-to-source transformation using the GHC API, which allowed us to dynamically load Haskell code to parse into astract syntax and generate abstract syntax elements to produce compilable Haskell code. The Template Haskell implementation is much cleaner.

5.1 Modifying Template Haskell

Our implementation relies on compile-time metaprogramming features of Template Haskell, which allows for the algorithmic construction of programs at compile time (Sheard and Peyton Jones 2002). With Template Haskell the programmer can programmatically construct abstract syntax for reification at compile time. The result is compiled and type-checked.

To obtain a list of the possible instantiations of a function, we have extended the GHC compiler to provide access to type class instance definitions in the type-checking environment. Our change involves extending the functionality of how the typechecker splices in Template Haskell code so that it can reveal compile-time type checking information. We add the following function, which returns a list of the possible instantiations of a given function:

```
instantiations :: TH.Name -> TcM [TH.Info]
```

This function does the following:

1. Looks up the signature associated with the function name.
2. Looks the instances associated with the class contexts in the signature.
3. Finds all possible instantiations of the function signature given the instances.

For instance, consider the following function signature:

```
f :: (Typeable a, Ord a) -> a -> a -> Bool
```

Suppose we had the following instances **Typeable Int**, **Typeable Bool**, and **Typeable String** for **Typeable** and the instances **Ord Int** , **Ord String** for **Ord**. The instantantiations of f that satisfy all requirements on type variable a are

```
f :: Int -> Int -> Bool,
f :: String -> String -> Bool.
```

We then wrap these in **ClassDynamic** constructors in the **inst** function, which returns the expression for reification.

5.2 Using a modified Template Haskell

With this extension we implement the **inst** function as shown in Figure 3 using an extension of Template Haskell. This function gets from the compile-time environment a list of Template Haskell expressions representing the functions with instantiated signatures based on the instances we have defined. We construct an expression

```
insts :: Name -> Q Exp
insts t = do
  instantiations <- instances t
  cd's <- listE (map proc instantiations)
  AppE (VarE (mkName "combineClassDynamic"))
       cd's
  where
  proc (VarI n ty _ _) =
    return $ AppE (VarE (mkName "toClassDyn"))
                  (SigE (VarE n) ty)
```

Figure 3. The **inst** function for getting instantiations from the type-checker environment.

that wraps each of the instantiations in a **ClassDynamic** data type and applies the function **combineClassDynamic** to yield a single representation. The compiler splices a reified version version of this expression into the program.

5.3 Typeable issues

In order to use the **ClassDynamic** constructor, type class variables must be declared in the context of Data.**Typeable**. This is so that we can call **typeOf** to examine type representations at runtime. We can programmatically declare as data types instances of Typeable without affecting existing code, as versions of GHC higher than 6.8 allow type class derivation to be declared after the initial type declaration.

We have also run into an issue with TypeableN: the current **ClassDynamic** type only handles instances of the **Typeable** class. The **Dynamic** library gets around this problem by having DynamicN ; since Haskell does not have kind polymorphism this may be the most desirable solution for generalizing the **ClassDynamic** library.

5.4 Static vs. dynamic conditional dispatch

An issue with this extension is the overhead required when the types are known at compile time. It is important to note that we do not intend the **dispatch** keyword to necessarily dynamic. Having a static conditional dispatch construct is quite useful: the **Set**, Print, and heterogenous list examples do not involve dynamic types. If we can statically examine the types, then the dispatches that can be resolved statically incur no overhead. A solution in the current framework with Template Haskell would involve inlining the calls to **dispatch** and examining the type environment.

6. ClassDynamic library

We implement dynamic dispatch using the functionality of a library we call **ClassDynamic**, which is based on Haskell's Data.**Dynamic** library (libraries@haskell.org May 2009), of which we show the interface in Figure 4. The novel contributions of the **ClassDynamic** library are

1. its support for sets of type class instances and
2. the **dispatch** function, which handles dynamic type class method dispatch.

We show the interface for our **ClassDynamic** module in Figure 5. The **ClassDynamic** library deviates from **Dynamic** in the following ways:

- Instead of having **dynApply** and **dynApp**, we have the **dispatch** function to handle dispatch among the possible specialized versions of a function.
- **ClassDynamic** has the **combineClassDynamic** function which takes a list of type [**ClassDynamic**] and combines the value-representation pairs to create a single **ClassDynamic** value if all values have the same kind, raising an exception otherwise.

```
module Data.Typeable
data Dynamic
toDyn :: Typeable a => a -> Dynamic
fromDyn :: Typeable a => Dynamic -> a -> a
fromDynamic :: Typeable a => Dynamic -> Maybe a
dynApply :: Dynamic -> Dynamic -> Maybe Dynamic
dynApp :: Dynamic -> Dynamic -> Dynamic
dynTypeRep :: Dynamic -> TypeRep
```

Figure 4. Interface for module Data.**Dynamic**.

```
module Data.Typeable

-- Analogous ClassDynamic functions.
data ClassDynamic
toClassDyn :: Typeable a => a -> ClassDynamic
fromClassDyn :: Typeable a => ClassDynamic -> a
             -> a
fromClassDynamic :: Typeable a
                 => ClassDynamic -> Maybe a
combineClassDynamic :: [ClassDynamic]
                    -> ClassDynamic
classDynTypeRep :: ClassDynamic -> TypeRep

-- Template Haskell function for accessing
     instantiations.
inst :: Name -> Q Exp
-- Conditional dispatch function.
dispatch :: ClassDynamic -> ClassDynamic
         -> ClassDynamic
```

Figure 5. Interface for module **ClassDynamic**.

6.1 Basics

The **ClassDynamic** data type stores collection of values with their associated type representations:

$$\textbf{ClassDynamic} = [\exists\tau.(typeRep, \tau)].$$

Since the existential is quantified over the right-hand side of the expression, the **ClassDynamic** constructor provides a universal abstraction over the type of the value. As with **Dynamic**, the presence of the typeRep allows us to crawl over **ClassDynamic** and discover the type τ.

The **ClassDynamic** library has the following constructors and destructors.

$$\texttt{toClassDynamic} :: \forall\tau.\tau \to \texttt{ClassDynamic}$$
$$\texttt{fromClassDyn} :: \forall\tau.\texttt{ClassDynamic} \to \tau \to \tau$$
$$\texttt{fromClassDynamic} :: \forall\tau.\texttt{ClassDynamic} \to \texttt{Maybe}\,\tau$$

The **fromClassDyn** function takes a default argument of type α and returns a value of the type α if one exists in the type representation; otherwise it returns the default argument. The **fromClassDynamic** function infers a type τ from the calling context instead of the default context, returning a value of type **Maybe** τ.

Claim 1. *We have the relationships:*

$$\frac{e :: \tau,\ d :: \tau}{\mathit{fromClassDyn}\,(\mathit{toClassDyn}\,e)\ d\ = e :: \tau}$$

$$\frac{e :: \tau',\ d :: \tau, \tau' \neq \tau}{\mathit{fromClassDyn}\,(\mathit{toClassDyn}\,e)\ d\ = d :: \tau}$$

If τ is the type from the calling context, we have:

$$\frac{e :: \tau}{\mathit{fromClassDynamic}\ (\mathit{toClassDyn}\, e) = \mathit{Just}\, e}$$

$$\frac{e :: \tau,\ \tau \neq \tau'}{\mathit{fromClassDynamic}\, (\mathit{toClassDyn}\, e) = \mathit{Nothing}}$$

Since Haskell values must be associated unambiguously associated with some type in in the static type environment, the length of the ClassDynamicInst list corresponding to e's representation must have a single element. Thus in the above definitions there is a unique e such that $e \in \ell_i$, where ℓ_i is the list of instances in the polymorphic representation. Both **fromClassDyn** and **fromClassDynamic** must return the unique element, which is a representation of e.

6.2 Class-based dispatch

A key feature of this library is the function

```
dispatch :: ClassDynamic -> ClassDynamic
         -> ClassDynamic
```

which takes a representation of a function `f :: t -> u` and a representation of a value `x :: t'` and returns the result **Just** `((f x) :: u)` if `t` is the same as `t'`, as shown in Figure 6. This is analogus to **dynApp** in the **Dynamic** library but handles dispatching for *all* versions of a function with appropriate types. We show the body of **dispatch** in Figure 7. The **dispatch** function takes each of the type representations of the function and compares it to each of the type representations of the arguments, adding each of the valid results to the set of type representations for the result. Like **dynApply** in the DynamicLibrary, we call unsafeCoerce once we determine we have a value of the appropriate type. We show some rules for **dispatch** in Figure 6.

Note that `t` and `u` can be polymorphic, as each **ClassDynamic** can consist of a list of possible instances. To handle this, we try each of the instance combinations in order to generate the list of instances for the resulting **ClassDynamic** value.

In the programs we get the instances using the function

```
inst :: Name -> Q Exp
```

which returns us a list of type **ClassDynamic**.

The **dispatch** operator, when used with **inst** should exhibit the same behavior as static dispatch, since **inst** returns a representation of all possible instances. Since the **dispatch** function refines the possible types as the function is applied to its arguments, **dispatch** should have the same semantics as in the case of static disptch, except instead of raising a type error it will return a value with an empty set of type representations.

6.3 Multiple representations and casting back

Claim 2. *A fully applied function contains multiple type representations only when the initial function has a type variable for the type of the result.*

We get our type representations from all instantiations of a function signature; these instantiations are unique. Since each argument of the function must be a concrete, monomorphic type, the only way to get multiple type representations when a function is fully applied is if the function signature has a type variable in the result. An example of such a function is

```
read :: (Read a) => String -> a
```

In the case of such functions, the cast back out with **fromClassDyn** or **fromClassDynamic** will only return a value of the appropriate type.

Consider the class declaration

```
dispatch :: ClassDynamic -> ClassDynamic
         -> ClassDynamic
dispatch (ClassDynamic fnInsts)
         (ClassDynamic argInsts) =
  let tryApply (ClassDynamicInst t1 f)
               (ClassDynamicInst t2 x) =
        case funResultTy t1 t2 of
          Just t3 ->
            [ClassDynamicInst
              t3 ((unsafeCoerce f) x)]
          Nothing -> []
      applyFnToArgs f =
        foldr (++) []
              (List.map (tryApply f) argInsts)
      appliedInsts =
        foldr (++) []
              (List.map applyFnToArgs fnInsts)
  in ClassDynamic appliedInsts
```

Figure 7. Source for for **dispatch**.

```
class C a where
  f :: Int -> a -> String
```

When we dispatch `f` on an integer, we will get back a **ClassDynamic** value containing a function of type `a -> String` for all instances `a` of the class `C`.

The function

```
fromClassDyn :: (Typeable a)
             => ClassDynamic -> a -> a
```

casts back to an arbitrary element in the type representation. If there is only one possible type, then this function will exhibit the correct behavior. We show that this is the case with fully applied functions. (For functions, we can use **dispatch** to perform the appropriate function application, so this is not a problem.)

7. Related work

Much work has been done with using *dynamic types* in a statically typed language (Abadi et al. 1989, 1994; Weirich 2000). Dynamic typing has allowed for the type analysis necessary for the Scrap Your Boilerplate approach to generic programming. In *Scrap Your Boilerplate*, Lämmel and Peyton Jones present a design pattern for generic programming that uses type-safe cast for type analysis when generating structure-traversing boilerplate (Lämmel and Peyton Jones 2003).

As we mentioned before, there has been desire to make type classes more flexible. We described in the examples section Haskell Wiki's proposed way of defining a default behavior for non-instances of a type class (Keslyov and Jones April 2008). The HLIST library supports strongly typed heterogenous collections: they point out that it is important to have guarantees on the elements in heterogenous lists and provide a way for doing so (Kiselyov et al. 2004). On the Haskell mailing list, Hal Daume posted a method for simulating class-based dynamic dispatch using existential types (Daume March 2003).

Somewhat related work in the vein of making ad-hoc polymorphism more flexible and usable is the work of Vytioniotis et. al. on $\lambda_{\mathcal{L}}$. $\lambda_{\mathcal{L}}$ combines the name-based type class mechanism for ad hoc polymorphism with polymorphism base on type structure (Vytiniotis et al. 2005). They observe that there are two common ways of defining ad-hoc polymorphic operations: inductively on the structure of types using *intensional type analysis* or over sets of types. Intensional analysis reduces structure-related boilerplate but is closed to extension by user-defined types. Type classes are open, but they may require tedious specialized boilerplate for

$$\frac{f :: \tau \to \mu, x :: \tau}{\texttt{dispatch (toClassDyn } f :: \tau \to \mu) \texttt{ (toClassDyn } x :: \tau) = \texttt{Just (toClassDyn } (f\ x) :: \mu)} \quad (1)$$

$$\frac{f :: \tau \to \mu, x :: \tau', \tau \neq \tau'}{\texttt{dispatch (toClassDyn } f :: \tau \to \mu) \texttt{ (toClassDyn } x :: \tau) = \texttt{Nothing}} \quad (2)$$

Figure 6. **dispatch** rules.

new types. $\lambda_{\mathcal{L}}$ combines both forms of polymorphism with a **typecase** operator for structural type analysis that is extensible to new user-defined types; the language has a first-class map from labels to expressions associated with the branches for **typecase**.

8. Conclusion

In this paper we have described a mechanism for function dispatch based on the type class memberships of a value's type. This allows us to base control flow on type class membership and provides more support for programming with dynamic types. We have

1. Proposed a mechanism for class-based conditional dispatch in the form of the **dispatch** operator.
2. Described an implementation within the current GHC framework using a proposed extension to Template Haskell and our **ClassDynamic** library.

We also speculate that this mechanism would be quite useful if it were more integrated with the Glasgow Haskell compiler.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–227, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: http://doi.acm.org/10.1145/75277.75296.

Martn Abadi, Luca Cardelli, Benjamin Pierce, Didier Rmy, and Robert W. Taylor. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5:92–103, 1994.

Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0262011530.

Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002. ISBN 1-58113-487-8. doi: http://doi.acm.org/10.1145/581478.581494.

Hal Daume. *Dynamic Dispatch on the Class of a Type*, March 2003. URL `http://okmij.org/ftp/Haskell/types.html`.

Mark P. Jones. A theory of qualified types. In *ESOP'92: Symposium proceedings on 4th European symposium on programming*, pages 287–306, London, UK, 1992. Springer-Verlag. ISBN 0387552537. URL `http://portal.acm.org/citation.cfm?id=145097`.

O. Keslyov and S. Peyton Jones. *Choosing a type-class instance based on the context*, April 2008. URL `http://haskell.org/haskellwiki/GHC/AdvancedOverlap`.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017488.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3): 26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

libraries@haskell.org. base-4.1.0.0: Basic libraries: Data.dynamic, May 2009. URL `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Dynamic.html`.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *In Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI*, pages 13–24, 2005.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989. URL `citeseer.ist.psu.edu/wadler88how.html`.

Stephanie Weirich. Type-safe cast: Functional pearl. pages 58–67, September 2000. URL `citeseer.ist.psu.edu/weirich00typesafe.html`.