Safe Dynamic Dispatch,

or How to Pickle with Class

A thesis presented

by

Jean Yang

to

Computer Science

in partial fulfillment of the honors requirement

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 2, 2008

Contents

Chapter 1

The dangers of method dispatch

> *If debugging is the process of removing bugs, then programming must be the process of putting them in. -Edsger Dijkstra*

In languages without compile-time type-checking, bugs may lie dormant for months until their branch of the code is executed. Though it is best to catch errors as early as possible, programmers often choose these dynamically typed languages over ones with compile-time correctness guarantees.

Dare we conclude that programmers are risk-loving? Because compile-time type-checking necessarily approximates run-time behavior, static type systems provide correctness guarantees only at the cost of ruling out programs that could be correct. When forced to take sides between static correctness and dynamic flexibility, the programmer may choose the latter.

In this thesis I present a favorable compromise between static and dynamic mechanisms. My investigation focuses on language constructs for grouping types by functionality. Careless coding and lack of language support frequently combine to cause the error of attempting to dispatch a method `f` that has not been defined for some object `obj` of type τ. Determining when it is safe to call `f` on `obj` requires knowing the types on which `f` has been defined. Current language support for doing this falls short, sacrificing either safety or flexibility.

I show that extending Haskell's static type class construct provides us with safe dynamic dispatch of type class methods. Haskell provides type classes for grouping types by functionality, but the static resolution prevents method dispatch when the types are not known until run time. I present a new library, `ClassDynamic`, and a source-to-source translation that support dynamic type class method dispatch.

In the remainder of this chapter, I examine the shortcomings of current language frameworks for method dispatch. In the rest of this thesis I give context to my solution (Chapter 2), describe the solution and its applications (Chapter 3), provide the implementation details (Chapter 4), and convince the reader of the importance of continued work in this area (Chapter 5).

1.1 Expressiveness or safety?

In this section I examine the safety and flexibility tradeoffs with Java interfaces and Haskell type classes. Neither Java nor Haskell, two of the few languages with support for grouping types by functionality, provides a desirable solution.

1.1.1 The dangerous dynamics of Java interfaces

Java provides the interface construct for encapsulating groups of related methods. A Java interface defines a set of methods required for objects implementing the interface.

I could create the Java interface `Picklable` to represent objects on which we can dispatch the serialization method `pickle`:

```
public interface Picklable {
    public String pickle();
}
```

A class providing an implementation must define the required function `pickle` and have the heading

`public class` *className* `implements Picklable`.

Please refer to Figure 1.1 for two classes implementing the interface.

Figure 1.2 demonstrates how we can use interfaces for determining which methods we can call on a given object. The `printPickledValue` takes an object, casts it to an object of type `Picklable` if we have an appropriate instance of the `Picklable` class, and prints the result of calling `pickle` on the object[1]. The static `instanceof` operator allows us to check whether an object has been declared as a subclass of the given class. Querying `instanceof` for an interface can tell us whether a class implements the interface. When the class is not known until run time, we can use the dynamically dispatched `isInstance` method, which is defined for all Java classes, to check whether we have an appropriate subclass. Changing the check from `(obj instanceof Picklable)` to `(obj.isInstance(Picklable))` would make the check dynamic.

There are many points of dynamic failure. The program will throw the `IllegalAccessException` if `obj` has primitive type. The program could also raise the `ClassCastException` for many reasons: if the user provided a class that does not implement the `Picklable` interface; if there is no public, no-argument constructor; if `c` is a null reference. These exceptions create a lose-lose situation: careless error catching allows dynamic exceptions; careful error handling causes bloated code, which obscures meaning and impedes the reader from reasoning about it easily.

[1] The reader may be inclined to ask why we did not just restrict the argument type to Picklable instead of Object. I provide two reasons: 1) this works better for this example and 2) perhaps we are getting an object from the network and can make not static guarantees about its type.

```
public class UserObject1 implements Picklable {
    String userValue1;

    public String pickle() {
        return userValue1;
    }
}

public class UserObject2 implements Picklable {
    String userValue2;

    public String pickle() {
        return userValue2;
    }
}
```

Figure 1.1: Classes implementing the `Picklable` interface.

```
void printPickledValue(Object obj) {
    if (obj instanceof Picklable) {
        Picklable p = (Picklable)obj;
        String s = p.pickle();
        System.out.println(s);
    } else {
        System.err.printf("Unable to perform request.");
    }
}
```

Figure 1.2: Classes implementing the `Picklable` interface.

```
class Picklable a where
  pickle :: a -> String

data UserObject1 = UserObject1 { userValue1 :: String }
instance Picklable (UserObject1 uVal) where
  pickle v = uVal

data UserObject2 = UserObject2 { userValue2 :: String }
instance Picklable (UserObject2 uVal) where
  pickle v = uVal

printPickledValue :: (Pickle a) => a -> IO ()
printPickledValue obj = putStrLn (pickle obj)
```

Figure 1.3: Haskell version of the `Picklable` library. It declares the new user defined data types `UserObject1` and `UserObject2`, each with its own value field.

1.1.2 Haskell's safer but limiting type classes

Haskell's statically-checked type classes eliminate the need for verbose exception-handling. Type class declarations specify an interface of methods and their type signatures; instances of type classes must implement the required methods.

Please see Figure 1.3 for the Haskell version of the previous example. This code defines a type class `Picklable` of types for which the `pickle` method is defined. The `(Picklable a)` constraint on the polymorphic argument of the `pickle` function allows the type checker to rule out dispatches to inappropriately-typed values at compile time.

Unfortunately, there is no Haskell mechanism that allows us to pass `pickle` a type that is not known to be an instance of `Picklable` at compile time. To achieve the equivalent functionality of `isInstance` in Haskell, we must go through the instances of the `Picklable` class to see whether we have a value of an appropriate type (Figure 1.4). I will explain the `Dynamic` data type and the `Maybe` constructor in more detail in the next chapter. The take-away point is that this mechanism for dispatching type class methods is undesirable, as it produces unnecessarily long code about which it is difficult to reason. Also, adding an instance of the `Picklable` class requires adding an additional case to `printPickledValue`.

1.2 A safer expressiveness

In this thesis, I show that we can support safe dynamic dispatch in Haskell with an extension that does not affect existing code. This extension allows us to specify method calls that discriminate

```
printPickledValue :: Dynamic -> IO ()
printPickledValue v =
  case fromDynamic v of
    Just (j :: UserObject1) -> putStrLn (pickle j)
    Nothing ->
      case fromDynamic v of
        Just (p :: UserObject2) -> putStrLn (pickle j)
        Nothing -> putStrLn "Unable to perform request."
```

Figure 1.4: Type case required for calling `pickle` on dynamic values.

based on whether we have an appropriate instance of some type class. To support this functionality, I have implemented a library `ClassDynamic` that contains a function for performing dynamic dispatch. I have also implemented a source-to-source translation that generates 1) a run-time dictionary mapping method names to instance definitions and 2) dynamic dispatch code for handling methods with variable result types.

Chapter 2

A brief history of typing

Strong typing is for people with weak memories. -Unknown

This chapter begins with an overview of type systems and ends with a discussion of Haskell's polymorphic type system and related work on type reflection and dynamic types.

2.1 Type systems and their guarantees

To motivate the existence of static type systems, let us step into a world without static checks. In Python, a dynamically checked language, there are few language abstractions preventing programmer error from causing run-time errors. Please refer to Figure 2.1 for a transcript of such a program. This program appears to run normally until the user enters 52, at which point it halts upon looking up the function `undefinedFunction`, which has apparently not been defined.

The source code of the program (Figure 2.2) reveals that the program was just waiting for an error to occur. There is a loop that prompts the user for integers, printing "No error yet!" until the user enters a number greater than or equal to 50. Because Python performs no checks on code branches until execution, careless errors such as calls to undefined functions can go undetected for months until the particular branch of code is executed.

Many of these errors can be ruled out at compile time. The desire for language prevention of programmer error has led to the rise of languages with stronger static guarantees. In many languages, static type systems provide a framework for the programmer to communicate expectations about the behavior of the code to machine and human readers. When I use the term *type-checking*, I refer to the compile time resolution of typing constraints. Type-checking can be based on explicit programmer-supplied annotations or involve type inference. Checking for constraint satisfaction is not limited to compile time: I use the term `static` to refer to what occurs at compile time and `dynamic` to refer to what occurs at run time. Static type systems are necessarily conservative: while they can prove the absence of certain bad program behaviors, they may reject programs

```
Please enter an integer: 6
No error yet!
Please enter an integer: 7
No error yet!
Please enter an integer: 42
No error yet!
Please enter an integer: 52
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    print undefinedFunction(x)
NameError: name 'undefinedFunction' is not defined
```

Figure 2.1: Transcript of a Python that runs until the user enters 52, at which point it encounters a run-time error.

```
while (True):
  x = int(raw_input("Please enter an integer: "))
  if x < 50:
    print 'No error yet!'
  else:
    print undefinedFunction(x)
```

Figure 2.2: Source code of Python program with run-time error.

that actually behave well at run time. For instance, most type-checkers will reject at compile time program like

```
if <true> then 42 else <ill-typed expression>.
```

Type systems can enforce type safety and type abstractions statically or dynamically. In a *type-safe* language, the type checker prevents primitives from being applied to incorrect argument types. A basic type checker can check that the programmer does not pass a string when an integer argument is expected. Languages supporting type abstractions also make it possible to extend primitive types with new type constructors. This allows the programmer to take advantage of type-checking mechanisms to enforce constraints hiding details of an abstract type's implementation. For instance, in a language where integers but not boolean values are defined, the programmer can either encode boolean values as the integers 0 and 1 or create a new data type. Doing the latter will allow the type checker to rule out the case in which an undefined value (i.e., 42) is used as a boolean.

2.2 Polymorphism comes in many forms

Creating type systems that allow *polymorphic* functions, which are defined for values of multiple types, is a difficult problem. When all values must have a type, how do we express and compute with a data structure that contain values of any type? In considering such problems we differentiate polymorphism into two categories, parametric and ad hoc.

Parametric polymorphism describes functions that exhibit the same behavior on all types and is useful because it allows programs to focus on algorithms rather than implementation details. A prototypical parametric polymorphic system is System F, also known as the *polymorphic lambda calculus*, discovered independently by logician Jean-Yves Girard (1972) and computer scientist John Reynolds (1974). System F allows for function definitions with quantified types such as

$$f : \forall X.(X \to X) \to X \to X.$$

The type signature says that f, defined on all types X, takes a function of type $X \to X$, a value of type X, and returns a value of type X. This type signature is revealing: the only action one could do with such requirements is to apply the first argument to the second. While this is theoretically beautiful, it becomes tedious to explicitly quantify and instantiate type signatures. ML, a descendent of System F, uses the Damas-Milner type inference algorithm to determine whether a program is type-safe in the absence of explicit type annotations [Pie02]. Haskell uses a variant of this ML's type system and type inference. While this type system is expressive enough to allow explicit and implicit quantifications, there are certain limitations. For instance, the type checker requires that all values be assigned a type at compile time.

Ad hoc, or *overloading*, polymorphism refers to the definition of a function that exhibits different behaviors on different data types. For instance, '+' could be defined to perform arithmetic addition

on integers but concatenation of strings. There is an inherent ambiguity associated with ad hoc polymorphism owing to the question of which of many actual functions/methods to invoke when the names are overloaded. Haskell's type classes provide a flexible and powerful form of overloading polymorphism.

Compiling with polymorphism is especially difficult because functions can have arguments of unknown type, functions can create record values whose types are unknown, and it is difficult to decide what code to generate when compiling an ad hoc polymorphic operation with arguments of variable type. In compiling ad hoc polymorphic programs, it is useful to use *intensional polymorphism*, which permits examination of run-time types [HM94]. The extension I propose relies on the examination of run-time types.

2.3 Key Haskell constructs

Here I focus on two of Haskell's language constructs, type classes and monads. Type classes provide a nice framework for ad hoc polymorphism and are the basis of the extension proposed in this thesis. Monads provide a useful pattern for encapsulating computation.

2.3.1 Notation

Before we begin, let us discuss how to interpret Haskell's type annotations. Haskell requires type signatures for type class method declarations and allows type signature annotations for function definitions. The type signature

```
f ::  Int -> String
```

represents a function that takes a single integer argument and return a string. An example of a full application is

```
f (42 ::  Int).
```

We need to attach a type annotation to `42` to disambiguate whether it is an integer- or real-valued number. A function taking two arguments, an integer and a boolean, and returning a string would have the signature

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

We write the argument types this way because the function can also be partially applied: it is a function that, when given an integer, returns a function of type `Bool -> String`.

More formally, a type signature has the form

$$fn :: [cx \Rightarrow]\tau,$$

where fn is the function name, cx is the optional context, and τ is the type of the function. The context consists of one or more type class constraints specifying the expected behavior of the

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

Figure 2.3: Haskell declaration of `Ord` type class [Com08a].

arguments. The type can be a simple type, a type variable, an arrow type representing a function, or the result of a type constructor application. Informally,

$$\tau := \tau \mid \alpha \mid \tau \rightarrow \tau \mid \tau_c\ \tau.$$

An example of a simple type is `Int`; an example function type is `Int -> String`, a function that takes an `Int` argument and returns a `String`; an applied type constructor is `[Int]`, which is `[]` (the list constructor) applied to `Int`.

The type $\tau \rightarrow \tau$ represents a higher-order type. A *higher-order* type, also called an *arrow* type, is one representing the type of a function. For instance, the anonymous function $(\lambda x.x + 1)$, which takes an argument and returns the result of adding 1 to it, could have type $Int \rightarrow Int$, a function type that takes an integer as an argument and returns an integer result.

The type $\tau_c\ \tau$ represents the application of a higher-kinded type. A *kind* is the type of a type constructor. We write $\star$ for the kind of a type, such as `Int`, and $\star \rightarrow \star$ for the kind of a type constructor, such as `Maybe` or `[]` (the list constructor), which takes a type and returns a type. For instance, the type `[Int]`, representing a list of integers, consists of the higher-kinded type constructor `[]`$:: \star \rightarrow \star$ applied to `Int`$:: \star$.

2.3.2 Type classes

Type classes group types by the functions that have been defined for them. The type signature

```
f ::  Ord a => Int -> a -> String
```

allows the second argument to be polymorphic within the `Ord` type class. This means that the second argument can now be of any type for which an instance has been defined for the `Ord` class. The `Ord` class (Figure 2.3) contains types for which comparison functions have been defined.

A class declaration introduces a new class and the operations associated with it. It generally has the form:

`class` cx `=>` u `where` [*declarations*]

When declaring a type class we declare new class methods, the optional fixity[1] of the class methods, and optional default class methods. Class methods live in the top level namespace with variable bindings and field names, meaning within a module, methods must have unique names [P+03].

An instance of a class C with superclass cx and type variable u is introduced with the following declaration:

$$\texttt{instance}\ cx'\ \texttt{=>}\ C\ (T\ u_1 \ldots u_k)\ \texttt{where}\ \{d\ \}$$

where $k \geq 0$ and the type $(T\ u_1 \ldots u_k)$ must take on the form of a type constructor T applied to simple (monomorphic) type variables $u_1, \ldots, u_k$. Such polymorphic instance declarations are forbidden:

```
instance C a where ...
instance C (Int,a) where ...
```

An instance declaration has the following static restrictions: 1) a type may be declared as an instance at most once in a program, 2) the class and type must share a kind, and 3) the type variables must satisfy the appropriate constraints [P+03].

The following code shows an instance declaration of the `Ord` class [Com08a]:

```
instance Ord Char where
  c <= c'          =  fromEnum c <= fromEnum c'
```

The `fromEnum` function converts a `Char` to an integer; the `<=` on the right hand side is invoking the integer comparison operation. Type classes are open: the programmer can easily add new instance definitions.

2.3.3 Monads

Since Haskell is meant to be a purely functional language, it has a curious way of handling side effects. *Side effects* are computations such as input/output and mutation with the property that the number of times we run them affects the output of the program. For instance, repeatedly running a procedure that prints the number "1" will output the unary representation of the number of times it is run. A *pure* computation has no side effects. We want functions to be pure because it is easier to reason about them for correctness checking and optimization.

To support computations with effects, Haskell uses the *monad* abstraction. Monads allow us to build delayed computations that are performed at run time. Since the computations are delayed, the monadic expressions are pure. To perform I/O within our function `f` from before, we could change the signature to

```
f_monadic ::  Ord a => Int -> a -> IO String,
```

[1]The *fixity* of a function `f` describes how `f` is to be called. If `f` is *prefix*, we call it before the arguments: i.e., given some argument `v`, we have the call `f v`. If `f` is *infix*, we call it between the arguments: i.e., given arguments `v1` and `v2`, we have the call `v1 f v2`.

which lifts the computation into the `IO` monad. The function `f_monadic` returns a delayed expression that, when run, returns a `String`. Running `f` performs a side effect, but the abstraction `IO String` prevents us from actually running the command.

Once we have entered a monadic computation, we can never get out: everything occurring after a delayed computation must also be delayed. Fortunately, we can build a whole program as a series of sequenced monadic commands. For instance, if we have

```
readInt ::  Int -> IO Int,
```

which reads and returns an integer from standard input and

```
writeInt ::  Int -> IO (),
```

which takes an integer and returns a computation printing the integer to standard output, we can compose `readInt` and `writeInt` using the command

```
bind ::  IO Int -> (Int -> IO ()) -> IO (),
```

which gives us the sequential composition

```
bind readInt writeInt,
```

This function reads an integer and then prints it out to screen.

To use a monad, we must have ways of lifting into the monad and sequencing monadic operations. Formally, a monad consists of a triple $(M, return, \texttt{>>=})$, which represents a type constructor M and the operations on M:

$$return :: \forall\alpha.\alpha \to M\ \alpha$$
$$\texttt{(>>=)} :: \forall\alpha,\beta.M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta$$

The `>>=` function is the generalization of `bind`. Haskell's `Control.Monad` class defines a type class framework for creating monads and performing operations within them [Com08b].

The `Maybe` monad provides a nice way of handling errors. The `Maybe` monad has two constructors, `Just` α and `Nothing`. If a computation could fail, lifting the computation into the `Maybe` monad allows us to return `Just (r ::  a)`, where `r` is the result of some type `a`, or `Nothing`. Suppose our previous function `f` can only return a string if the input satisfies some properties. Rewriting the signature of `f` as

```
f_optional ::  Ord a => Int -> a -> Maybe String
```

allows us to return `Just (s ::  String)` upon success and `Nothing` otherwise [Com08a].

2.4 Related work

The `Typeable` library supports type reflection by providing a type class for types that can admit a representation at run time. Type reflection allows us to have dynamic types.

2.4.1 Typeable type reflection

Haskell's `Typeable` library defines the `Typeable` type class of types that can admit a run-time representation [BS02]:

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

`TypeRep` is a data constructor containing a runtime representation of an object's type. The `Typeable` library supports the manipulation of type representations and includes the following functions [Com08a]:

```
mkAppTy :: TypeRep -> TypeRep -> TypeRep
mkFunTy :: TypeRep -> TypeRep -> TypeRep
funResultTy :: TypeRep -> TypeRep -> Maybe TypeRep
```

The functions `mkAppTy` and `mkFunTy` allow for the creation of `TypeRep` objects. The `mkAppTy` function takes a type constructor and a type to create a type representation; `mkFunTy` takes two types τ_1, τ_2 and creates an arrow type representation of $\tau_1 \rightarrow \tau_2$. For instance, to build a run-time representation of the type `Int -> String` we would call

`mkFunTy` [*typerep* `Int`] [*typerep* `String`].

There are also dual functions that take apart a type representation: `funResultTy` takes a representation of $\tau \rightarrow \mu$ and a representation of τ' and returns a representation type μ if $\tau \equiv \tau'$.

`Typeable` also gives us the type-safe cast function[Wei00]

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

which allows us to examine the type representations of `a` and `b` and return the result of the cast if and only if `a` and `b` are structurally equivalent. In *Scrap Your Boilerplate*, Lämmel and Peyton Jones present a design pattern that uses type-safe cast for generating the boilerplate code necessary for traversing data structures [LP03]. This method allows for the creation of *generic*, structure-agnostic algorithms. In the next chapter, we will use generic traversal in a motivating example.

```
listToString :: [Dynamic] -> String
listToString nil = ""
listToString h::rest =
  (case fromDynamic d of
    Just (i :: Int) -> pickle i
    Nothing ->
      case fromDynamic d of
        Just (s :: String) -> pickle s
        Nothing ->
          case fromDynamic (b :: Bool) -> pickle b
          Nothing -> "<list element>") ++ (listToString rest)
```

Figure 2.4: Case analysis on types for printing list of type `[Dynamic]`. For the purposes of this example, assume that instances of `Picklable` have been defined for `Int`, `String`, and `Bool`.

2.4.2 Dynamic types

The `Dynamic` library, built using the `Typeable` class, provides the `Dynamic` data type, which stores a value of any type with its type representation. `Dynamic` allows us to have dynamically typed values that escape the static type checker. Dynamic types are useful for representing objects whose types are not known at compile time. Dynamic typing is important when doing persistent programming, where values of any type are stored and retrieved from storage, and distributed programming, where remote programs exchange data and code [SSP97].

The `Dynamic` library has functions that allow the programmer to convert to and from a `Dynamic` value, get the type representation of a `Dynamic` value, and apply `Dynamic` functions to `Dynamic` values [Com08a]. In particular, the function

```
toDyn :: Typeable a => a -> Dynamic
```

takes a value of any type `a` and creates a `Dynamic` value that stores the value with its type representation. In the context of a type `a`, the function

```
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

takes a `Dynamic` value and returns `Just (v :: a)`, where `v` is the value with type `a`, or `Nothing`.

Because `Dynamic` allows us to subvert the type checker, we can use `Dynamic` to create heterogeneous lists. Since all Haskell values must have a concrete type at run time, we cannot have a list of type $[\alpha]$, where α is of variable type. `Dynamic` provides a way of getting around this: we can have a list of type `[Dynamic]`, as the type `Dynamic` is of monomorphic type. Because the `Dynamic` constructor retains the type representation, we can examine and reify the types of the list values.

Consider the list definition

```
dynList = [toDyn (42 ::  Int), toDyn "Hello World", toDyn True].
```

In Figure 2.4 I show code to traverse a list for printing. Printing requires case analysis on the types because the call to `fromDynamic` needs to be assigned a type at compile time. Because the case variables have appropriate compile-time types, we can call `pickle`.

There are two drawbacks to this approach: 1) the type case can get quite extensive and 2) the function definition is not open to adding new types to the list. For instance, if someone were to add a real-valued number to the list, I would need to add a case to accommodate `Float`. Such a function would not do well in a library, as it cannot accommodate any user-defined data types. In the next chapter I describe a more preferable mechanism.

Chapter 3

Using dynamic type class method dispatch

One of my most productive days was throwing away 1000 lines of code. -Ken Thompson

In this chapter I provide a more complete specification of the problem, describe the solution, and demonstrate that the solution is useful.

3.1 Dynamic dispatch for type class methods

Recall from Chapter 1 discussion of why it would be useful to dispatch type class methods on values of dynamic type. To use the type class functions with values of type `Dynamic`, we need to perform a (perhaps extensive) case analysis to cast the values at each of the possible types. It would be convenient to be able to cast a dynamic value as an element of a type class:

```
printPickledValue :: Dynamic -> IO ()
printPickledValue v =
  case fromDynamicClass Picklable v of
    Maybe (p :: (Picklable a) => a) -> putStrLn p
    Nothing -> putStrLn "Unable to perform request."
```

In this function we want `p` to have variable type `a`, where `a` can be any instance of the `Picklable` class. Unfortunately, since Haskell requires that `p` have a concrete type at compile time, the Haskell type checker will not permit this. In the next sections I describe how we can achieve this functionality in the current Haskell framework.

```
dynamicDispatch :: Map String [Dynamic] -> String -> Dynamic ->
                   Maybe Dynamic
dynamicDispatch dictionary funName arg =
  let valTy = dynTypeRep arg
      matchType ty tyList = List.filter (applicableToTy ty) tyList
  in case (Map.lookup funName dictionary) :: (Maybe [Dynamic]) of
      Just dynFuns ->
        case (matchType valTy dynFuns) of
        fn:_ -> dynApply fn arg
        [] -> Nothing
      Nothing -> Nothing
```

Figure 3.1: Basic version of the dynamic dispatch function.

3.1.1 Fitting into the Haskell framework

The extension I propose provides the equivalent functionality as the cast by performing dynamic dispatch on the type class method. To perform dynamic dispatch, we need to know at run time the type class instances that have been defined and we need a way to represent methods. To address these issues I provide a way for constructing run-time dictionary mapping function names to typed instance declarations. For purposes of method lookup it is most sensible to represent method as strings containing their names.

Having the dictionary of instance declarations allows us to write a `dynamicDispatch` function to perform dynamic dispatch (Figure 3.1). function compares the type representation of the argument `arg` against the type representations of the dictionary functions, returning a `Dynamic` value containing the applied result if successful and `Nothing` otherwise.

The `dynamicDispatch` function allows us to write `printPickledValue` as the following function, where `PicklableInstances` represents a dictionary containing the defined instances of `Picklable`:

```
printPickledValue :: Dynamic -> IO ()
printPickledValue v =
  case dynamicDispatch PicklableInstances "pickle" v >>= fromDynamic of
    Maybe (s :: String) -> putStrLn s
    Nothing -> putStrLn "Unable to perform request."
```

The dictionary consists of pairs like

```
("pickle", [toDyn (c ::  Int -> String), toDyn (c ::  Bool -> String)]).
```

The list elements must be typed to specify which instance of the method should be invoked. For

```
class C2 a b where c2 :: a -> b -> String
instance C2 Int Int where c _ _ = "(Int, Int)"
instance C2 Int Bool where c _ _ = "(Int, Bool)"
```

Figure 3.2: Multi-argument type class method.

this purpose it is particularly convenient to use `Dynamic` constructor because 1) we want to construct a list of heterogeneous type and 2) we want to store the type representation for use in the `dynamicDispatch` function.

3.1.2 A more flexible Dynamic

While using `Dynamic` and `dynamicDispatch` works for a function like `pickle`, we encounter problems with type representation with multi-arguments methods. With a multi-argument method, we need the full application to determine unambiguously which method definition to invoke. Consider the definition of the `c2` function in Figure 3.2. The full application

```
(c (1 ::  Int) (2 ::  Int))
```

would yield the result `"(Int, Int)"`. The partial application

```
c (1 ::  Int)
```

yields a function that could be applied to a value of type `Int` or `Bool`. When returning the dynamically dispatched result we want to represent the type

```
[Int -> String, Bool -> String],
```

but the `Dynamic` data type only supports monomorphic representations.

As a solution I propose the `ClassDynamic` data type, which stores a list of type representations. The `ClassDynamic` library that I present in the next chapter has functions for manipulating values of type `ClassDynamic`. The library includes the `classDispatch`, the `ClassDynamic` version of `dynamicDispatch`.

3.1.3 Methods with variable result type

With this approach we still have an issue with methods with variable result types. Currently, the Haskell type checker requires the program to provide the context for disambiguating the method dispatch. Since this approach is suboptimal for dealing with values of dynamic type, I introduce a more flexible mechanism.

```
case classDispatch instanceDictionary "read" "True" >>= fromClassDynamic of
  Just (x :: Bool) -> putStrLn (show x)
  Nothing -> putStrLn "Couldn't process request."
```

Figure 3.3: Code for calling `classDispatch` on `read :: (Read) => String -> a`. Assume we have a dictionary `instanceDictionary`.

When calling the `Read` class method

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

we must provide enough context for the type checker to infer the type of the result. For example, we can make the call

`read "True" :: Bool`.

However, the type checker has no way of distinguishing this from the call

`read "True" :: Int`,

which it also allows because there is no way to statically determine whether the call will fail. Because of particular definition of `read`, this call will actually raise the run-time exception "no parse."

With dynamic dispatch we have the same ambiguity problem in a different form. The call

`classDispatch` [*dictionary*] `"read" "True"`

would give us `ClassDynamic` value representing all instance declarations of the `Read` class, for which `read` is a method. We must supply a type when calling `fromClassDynamic` to get back the corresponding value for the code in Figure 3.3. This code, which dispatches on the `read` function and converts the result back from a `ClassDynamic` value, would output "True." If we changed `(x :: Bool)` to have another type, such as `(x :: Int)`, we would get the run-time error "no parse."

This solution for methods of variable result type is unsatisfactory for dynamic dispatch: when receiving a arbitrary strings across the network, we may not be able to statically anticipate the result type of a call to `read`. In such situations it would be convenient to make a call like

`dispatchOn read`,

which returns a value of type `[ClassDynamic]`, the list of all values for which `read` could return a value.

I implement this functionality by constructing a specialized function for each type class method. Please refer to Figure 3.4 for the specialized `dispatch` function for `read`, which tries to call `read` for all instance types and returns the results for which the call did not raise an errors. This computation must be lifted into the `IO` monad because it handles exceptions. For dynamic dispatch of general functions with variable result type, I define a keyword `dispatchOn` which allows the call

`dispatchOn read "True"`,

translating it into the call

`dispatch_read "True"`.

I implement a source-to-source translation that generates the `dispatch_f` corresponding to the method `f`.

This solution generalizes for function with n arguments and variable result type. For a method `f`, `dispatch_f` must take the same number of arguments: we need to apply all arguments to determine the result type. When we have n arguments, we cast on n-tuples. For instance, for some method `doubleArgs` with 2 arguments we could construct a function like the following:

```
dispatch_doubleArgs :: ClassDynamic -> ClassDynamic -> IO [ClassDynamic]
dispatch_doubleArgs x0 x1
  case (fromClassDynamic x0, fromClassDynamic x1) of
    (Just (b0 :: type1), Just (b1 :: type2)) ->
      ...
```

We can generalize the function to return the results in any container. Using the `MonadPlus` class, which requires the definitions of the empty element `mzero` and an accumulation function `mplus`, we can generate a function with signature

`dispatch_read :: (MonadPlus m) => ClassDynamic -> IO (m ClassDynamic)`,

where we replace `++` with `mplus` and `[]` with `mzero`. Now we can get our matching results in the data structure of our choice.

3.2 Applications

These new mechanisms are useful not just for handling dynamic types, but also for increasing the flexibility of method dispatch. In this section I demonstrate the use of dynamic type class method dispatch for multiple dispatch, for reusing type class method code, for guarantees on the behaviors of methods, and for convenient interpretation of method names to methods.

```
dispatch_read :: ClassDynamic -> IO [ClassDynamic]
dispatch_read x0 =
  case fromClassDynamic x0 of
    Just (b0 :: String) ->
      do t1 <- E.catch (do return $! (read b0) :: Bool
                           return [toClassDyn $! (read b0 :: Bool)]
                 $ (\_ -> return [])
         t0 <- E.catch ...
                   ...
                     -- Int case
         return (t1 ++ t0)
    Nothing -> []
```

Figure 3.4: The `dispatchOn` function for `read`, which tries calling `read` on each of the instances. We make each call twice: the first is to see if we raise an exception; the second call to return the applied result occurs only if the first did not raise and exception. The `$` operator is a sequencing operator that eliminates the need to put parentheses around the expression to the right of the operator. The `!` operator forces the evaluation; otherwise, Haskell only evaluates an expression when it is used.

3.2.1 Flexible multiple dispatch

Haskell allows a rigid form of static multiple dispatch. Dynamic dispatch helps eliminate boilerplate code when performing multiple dispatch on dynamic values and increases the flexibility of dispatching on overloaded methods by providing a mechanism for returning all appropriate dispatch results.

Multiple dispatch describes a solution to determining which method to dispatch for an overloaded multi-argument method. *Single dispatch* refers to the situation where a single argument determines the dispatch; *multiple dispatch* refers to the situation where all arguments participate in the decision. Multiple dispatch avoids problems of asymmetry. If we had an implementation that determined the type of the result based on the type of the first argument, the expression $42_{Integer} + 43.1_{Real}$ would evaluate to 85 while the express $43.1_{Real} + 42_{Integer}$ would evaluate to 85.1. The asymmetry of single dispatch can cause behavior that makes no mathematical sense [Cha92].

There are two issues with current Haskell-style multiple dispatch: 1) we would need to specify all types and 2) it does not extend well to `Dynamic` values. Consider the multiply dispatched method parameterized by both argument types and by the return type in Figure 3.5. For the type checker to unambiguously determine the method dispatch, we would need add type annotations to each of our arguments:

```
m (43.1 ::  Float) (42 ::  Int) ::  Float.
```

To use this method with `Dynamic` arguments, we would need to perform a type case for each of

```
class Multi a b c where
  m :: a -> b -> c
instance Multi Int Int Int where
  m x y = x + y
instance Multi Float Int Int where
  m x y = y
instance Multi Float Int Float where
  m x y = x
```

Figure 3.5: Type class definition and instance declarations for multiple dispatch method.

```
processMultiDynamic :: Dynamic -> Dynamic -> IO ()
processMultiDynamic x y =
  case (fromClassDynamic x, fromClassDynamic y) of
    (Just (i0 :: Int), Just (i1 :: Int)) ->
      ...
        ...
```

Figure 3.6: Type case necessary for multiple dispatch on dynamic types.

the arguments, as shown in Figure 3.6. Matching on all types is undesirable because 1) it places unnecessary burden on the programmer, 2) it is difficult to reason about the correctness of the resulting code, and 3) the number of cases explodes combinatorially as the number of arguments grows.

The dynamic dispatch extension provides two options for performing multiple dispatch on `Dynamic` values: if we know the desired result type, we can use the `classDispatch` function. If we want all possible result types, we can use the `dispatchOn` function to try all possible result types. Please refer to Figure 3.7 for the code with `classDispatch`, which produces a result by sequencing operations within the `Maybe` monad. The function calls `classDispatch` to apply the function to the first argument, applies the result to the second argument, and calls the `pickle` method to print the result as a string.

The code using `dispatchOn` (Figure 3.8) shows that the extension also provides a more flexible mechanism for multiple dispatch by allowing us to expressing all possible results of a dispatch. The call `processValues (toPolyDyn "42") (toPolyDyn "24")` matches all methods and displays

```
<<[<<Int>>, <<Int>>, <<Float>>]>>.
```

Note that even when we know the types of the arguments, we can construct `ClassDynamic` values and call `classDispatch` in order to get all matching results.

```
processMultiValues :: ClassDynamic -> ClassDynamic -> IO ()
processMultiValues a b =
  let polyApp f = polyDynApply f y
      -- Sequence operations in Maybe monad to produce result.
      processResult r = polyDispatch functionMap "pickle" r >>= fromPolyDynamic
      r = polyDispatch functionMap "m" x >>= polyApp >>= processResult
  in
    case r of
      Just (s :: String) -> putStrLn s
      Nothing -> putStrLn "Couldn't process request."
```

Figure 3.7: Multiple dispatch dynamically with `classDispatch`. Assume that an instance of `pickle` has been defined for lists.

```
processMultiValues :: ClassDynamic -> ClassDynamic -> IO ()
processMultiValues a b =
  case dispatchOn m a b of
    _:_ -> putStrLn (pickle v)
    [] -> putStrLn "no results"
```

Figure 3.8: Multiple dispatch dynamically with `dispatchOn`.

3.2.2 Reusable type class code

Dynamic dispatch allows us to define type class methods in terms of other type class methods without an explicit constraint in the type signature.

This is useful is for writing generic functions. In Chapter 2, I mentioned that Lämmel and Peyton Jones show how to use type-safe cast to generate the boilerplate structure-traversing code. The result is incorporated into the `Data` type class, which aids the Haskell programmer in writing *generic*, parametrically polymorphic functions that perform the same action regardless of the type of the argument. Putting `Data` in the function's context guarantees that we can call structure-traversing methods on the argument.

For those wondering why type-specific code would be useful when defining a type-agnostic method, consider the following example. Suppose we wanted to write a generic serialization method

```
gpickle ::  (Data a) => a -> String
```

that recursively serializes the elements of some structure. Given our goals, it makes sense to write this in terms of method `pickle` from before. Unfortunately, there is no `Picklable` restriction on the polymorphic argument `a`, as `gpickle` should be able to serialize any structure. Thus we cannot `pickle` in the body of `gpickle` without extensive case analysis, as shown in Figure 3.9. With the extension, we could easily write `gpickle` in terms of `pickle`, as shown in Figure 3.10.

One may object that such functions are no longer generic, as their behavior depends on run-time type analysis. Though the functions are no longer type-agnostic, the specialized behavior occurs at the type class level rather than the type level. The `classDispatch` abstraction makes the code easier to reason about, as the only typed-based discrimination that occurs is whether the type has been defined as an instance of some type class.

3.2.3 Functional guarantees

With the extension we can also build frameworks for passing around computation with enforceable behavioral guarantees. In a Haskell framework for passing code around a network, we could use types to exclude potentially harmful computations such as disk I/O. Such guarantees would be useful for peer-to-peer networks and for applications like λbot, a command line tool written in Haskell that runs an interactive interpreter loop through web chat. λbot accepts Haskell code, runs it, and returns the serialized result. λbot rules out potentially harmful computation by rejecting computations with unserializable results. Since types lifted into monads have not been defined as serializable, λbot does not accept computations with side effects.

If we were accepting functions as arguments but wanted to restrict the function types to have the form

```
(Typeable a) => a -> String,
```

```
gpickle = ( \t ->
                "("
             ++ (case (cast t :: Maybe Int) of
                    Just i -> pickle i
                    Nothing ->
                      case (cast t :: Maybe Float) of
                         Just f -> pickle f
                         Nothing ->
                           case (cast t :: Maybe Bool) of
                             Just b -> pickle b
                             Nothing ->
                               ...
                               -- After we've listed all Picklable instances...
                               Nothing -> showConstr(toConstr t))
             ++ concat (gmapQ ((++) " " . gpickle) t)
             ++ ")"
          )
```

Figure 3.9: The `gpickle` function without the extension. The methods `showConstr` and `toConstr` are methods of the `Data` type class for manipulating a data type. The call `showConstr (toConstr t)` returns the string representation of t's constructor.

```
gpickle = ( \t ->
                "("
             ++ (case polyDispatch funDict "pickle" (toDyn t) of
                    Just x -> case fromClassDynamic x of
                      Just (s :: String) -> pickle s
                      Nothing -> showConstr(toConstr t)
                    Nothing -> showConstr (toConstr t))
             ++ concat (gmapQ ((++) " " . gpickle) t)
             ++ ")"
           )
```

Figure 3.10: The `gpickle` function in terms of `pickle`.

```
runFunction :: () Dynamic -> a -> IO ()
runFunction f x =
case classDispatch funDict "run" f of
  Just x -> case polyDynApply fn (toPolyDyn x) of
              Just polyR -> case fromClassDynamic polyR of
                              Just (r :: String) -> putStrnLn r
                              Nothing -> putStrLn "Wrong"
          Nothing -> putStrLn "Failed in application.\n"
  _ -> putStrLn "No valid function was provided.\n"
```

Figure 3.11: This function only runs functions of type `(Typeable a) => a -> String`.

we could write the following class:

```
class (Typeable a) => Run a where
  run :: (a -> String) -> a -> String
```

The first argument is a function of type `(a -> String)`, which takes any value that can admit a type representation and returns a string. The `String` return type prevents the function from lifting the computation into any monad.

We could use `Run` to make sure we only accept functions with types of a particular structure by defining the instances we are willing to accept. With `classDispatch`, we can easily write functions that only accept "appropriate" methods, as shown in Figure 3.11. This function requires the caller to declare the argument `f` as an instance of the class. We could make the following instance declaration:

```
instance Run Int where
  run :: (Int -> String) -> Int
  run f v = f v
```

The function applies `f` to `x`, yielding a string. With these declarations, a call to

```
runFunction (toPolyDyn (λ (x ::  Int) -> show (x + 10)))
            (42 ::  Int)
```

would yield the integer 52. A call to

```
runFunction (toPolyDyn (λ (x ::  Int) -> putStrLn (show (x + 10))))
            (42 ::  Int)
```

would yield the output "No valid function was provided."

```
class FunctionGroup a b where
  fun1 :: (Typeable a) => a -> IO ()
  fun2 :: (Typeable b) => b -> IO ()
instance FunctionGroup Int String where
  fun1 i = putStrLn (pickle i)
  fun2 s = putStrLn s
```

Figure 3.12: Type class for the purpose of dispatching methods from string representations.

```
classDispatch funDict "fun1" (toClassDyn (42 :: Int))
classDispatch funDict "fun2" (toClassDyn (42 :: Int))
classDispatch funDict "fun2" (toClassDyn "Hello World")
```

Figure 3.13: Example dynamic type class method dispatches, where `funDict` is the dictionary containing the instance definitions. Note that the second dispatch will return `Nothing` since the argument is of an incompatible type.

3.2.4 Strings to methods

Since we now have at run time a dictionary mapping strings to type class methods, we also have a convenient way to use strings to invoke type class methods. Currently, running methods whose names are passed in as strings is only possible through performing case analysis on the possible names.

Because we have a run-time dictionary of all type class method names, which live in the top level namespace, we can call

`classDispatch` [*dictionary*] [*method name*] [$first\ argument$].

For example, if we had a set of methods we might want to call, we could define something like the class in Figure 3.12. This allows us to make calls like the ones in Figure 3.13.

This mechanism provides a limited form of the functionality of Lisp/Python's `eval` function, which can take a string and evaluate it as code. Using `classDispatch` is much safer than using `eval`, as the result is lifted into the `Maybe` monad. The static type checker requires the programmer to consider both the `Just` case and the `Nothing` case when recovering the result. This leverages the type system to for static enforcement of proper error handling.

Chapter 4

Behind the scenes

Simplicity does not precede complexity, but follows it. -Alan Perlis

In this chapter I describe the implementation of the `ClassDynamic` library, the code generation for run-time dictionaries, and the code generation for type class methods with results of variable type. The implementation of is quite clean because the Glasgow Haskell Compiler (GHC) provides good abstractions: the `Typeable` library provides a good abstraction for type reflection and the GHC API library provides a nice interface for dynamically parsing Haskell source code into type-checked abstract syntax.

4.1 ClassDynamic library

The novel contributions of the `ClassDynamic` library are 1) its support for representing intersections of sets of type class instances and 2) the `classDispatch` function, which handles dynamic type class method dispatch. In this section I highlight the key functionality of the library; please refer to Appendix .1 for the complete library interface.

The `ClassDynamic` library defines the `ClassDynamic` data type, which stores a list of values with their respective type representations:

$$\texttt{ClassDynamic} = [\exists\tau.(v :: \tau,\ \mathit{typeRep}\ \tau)].$$

I use the existential quantifier to represent a list of values, each of which has some type. The existential quantifies only over each list item: the `ClassDynamic` data type itself does not leak any information about τ. The `typeRep` allows us to crawl over `ClassDynamic` to discover the identity of τ in order to use v at the type τ. The `ClassDynamic` data type can store handle higher-order types (i.e., `Int -> String`) and applications of higher-kinded types (i.e., `[Int]`).

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

$$\begin{aligned} \texttt{toClassDynamic} &:: \quad \forall\tau.\tau \to \texttt{ClassDynamic} \\ \texttt{fromClassDyn} &:: \quad \forall\tau.\texttt{ClassDynamic} \to \tau \to \texttt{Maybe}\ \tau \\ \texttt{fromClassDynamic} &:: \quad \forall\tau.\texttt{ClassDynamic} \to \texttt{Maybe}\ \tau \end{aligned}$$

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 function `fromClassDynamic` 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 following relationships*[1]*:*

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

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

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 are necessarily monomorphic, 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.

We have the `classDynApply` function for applying `ClassDynamic` functions to `ClassDynamic` arguments, defined as

```
classDynApply ::  ClassDynamic -> ClassDynamic -> Maybe ClassDynamic.
```

The function 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'`:

$$\frac{f :: \tau \to \mu, x :: \tau}{\texttt{classDynApply (toClassDyn}\ f :: \tau \to \mu)\ \ \texttt{(toClassDyn}\ x :: \tau) = \texttt{Just (toClassDyn}\ (f\ x) :: \mu)} \tag{4.1}$$

[1]The statements above the line represent the assumptions and the statement below the line represents the claim.

$$\frac{f :: \tau \to \mu, x :: \tau', \tau \neq \tau'}{\texttt{classDynApply (toClassDyn } f :: \tau \to \mu) \texttt{ (toClassDyn } x :: \tau) = \texttt{Nothing}} \tag{4.2}$$

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

The most important function is

```
classDispatch :: [ClassDictEntry] -> String ->
                 ClassDynamic -> Maybe ClassDynamic,
```

for dynamic type class method dispatch. The `classDispatch` function takes a dictionary, the name of a type class method, and the value to pass to the function. The data type `ClassDictEntry` stores a mapping from type class method names to a list of `ClassDynamic` value storing the method instance. I define it as the record type

```
data ClassDictEntry = CDE { name :: String, fns :: [ClassDynamic] }
```

Similar to the `dynamicDispatch` function in the previous chapter, `classDispatch` returns a `Maybe ClassDynamic` representing the function applied to all matching type class methods.

Because we can have a ClassDynamic value that stores multiple values for a concrete type τ, we also need to be able to return a list of all values of a given type.

Claim 2. *The functions* `fromClassDyn` *and* `fromClassDynamic` *evaluate nondeterministically. That is, given* $p ::$ `ClassDynamic` *and a type* τ, $p =$ `ClassDynamic` ℓ_i, *there may exist multiple* $e \in \ell_i$ *such that* $e :: \tau$.

Consider the class declaration

```
class C a where c :: a -> String
instance C Int where c _ = "Int"
instance C Float where c _ = "Float"
```

This is because if we had a `ClassDynamic` value `v` resulting from a call to `read` consisting of

```
[1 :: Int, 1.0 :: Float]
```

and called

```
classDispatch funDict "c" v,
```

the result would be a representation of

```
["Int" :: String, "Float" :: String].
```

Calling `toClassDyn` and `toClassDynamic` would yield only one of these; either one could be returned.

Because of the possible nondeterminism we also have the function

$$\texttt{fromClassDynamicAll} :: \forall \mu, \tau.(\texttt{MonadPlus}\ \mu) \Rightarrow \texttt{ClassDynamic} \rightarrow \mu\ [\tau],$$

which uses a monad to generalize over data structure constructors.

Another useful function is

$$\texttt{combineClassDynamic} :: \forall \mu, \tau.(\texttt{MonadPlus}\ \mu) \Rightarrow \mu\ \texttt{ClassDynamic} \rightarrow \texttt{Maybe ClassDynamic},$$

which takes a data structure of `ClassDynamic` values and combines them into a single `ClassDynamic` value if the values have the same kind. This function is especially useful for combining the list of type `[ClassDynamic]` we get from calling `dispatchOn`. For instance, we could have a call to `read` that returns a list representing

$$\texttt{[42 :: Int, 42.0 :: Float]}.$$

We can combine these two values into a single `ClassDynamic` value for continued computation.

Please refer to Appendix .2 for example uses of the `ClassDynamic` library.

4.2 Implementation

The GHC API allows us to both parse Haskell source code into abstract syntax and generate abstract syntax elements producing compilable source code. With the GHC API we can dynamically import libraries and dynamically load source code into parsed, type-checked abstract syntax [Com08b].

4.2.1 Crawling over the abstract syntax

Generating code for the dictionary and dispatch functions for unparameterized result types involves crawling over the abstract syntax tree (AST) for the necessary type class information and producing new abstract syntax for the produced modules. We can crawl over AST to collect the necessary information about type class declarations and instance declarations.

The program-supplied type class declaration and instance definition information provide us with all information we need for the extension. The standard class and instance declarations in Figure 4.1 provide enough information to determine that we have methods with the signatures

```
c :: Int -> Bool -> Int -> Int,
c :: Float -> String -> Float -> Int.
```

```
class (Typeable a, Typeable b) -> C a b where
  c :: a -> b -> a -> Int
instance C Int Bool where
  c _ _ _ = 0
instance C Float String where
  c _ _ _ = 1
```

Figure 4.1: Standard Haskell type class declaration and instance definitions.

For the source-to-source translation I store 1) a map from type class names to the type variables of type classes and the type schemes of the functions and 2) a map from method names to the types of the instance definitions. For the example above, we would associate the type class `C` with the variables `[a, b]` and the signature

`c :: a -> b -> a -> Int`.

We would store the that `c` has instance definitions for `[Int, Bool]` and `[Float, String]`. This information allows us to instantiate the method signatures to create a method name to instance map with pairs like

(`"c"`, `Int -> Bool -> Int -> Int`).

With such a map we can easily generate our dictionaries and `dispatch` functions. No type-checking information is necessary.

4.2.2 Generating code

Using the GHC API and the map of method names to method signatures, generating the dictionary as abstract syntax is straightforward. The GHC API has functionality that allows me to pretty-print the AST as code to a local file, which is by default `DynamicDispatch.hs`. I name the dictionary `instanceDictionary` and give it the type `[ClassDictEntry]`. A sample dictionary is as follows:

```
instanceDictionary =
  [CDE "c"
       [toClassDyn (c :: Int -> Bool -> Int -> Int,
        toClassDyn (c :: Float -> String -> Float -> Int)]]
```

We can also generate the `dispatch` functions using the map of method names to signatures. From the signatures we can derive the number of arguments, the types of each of each of the arguments, and the types of the intended results. Creating the case statements for a given function involves crawling over the instance definitions and their types. Again, we can use the pretty printer

to generate code from abstract syntax[1]. These functions also live in `DynamicDispatch.hs` by default.

4.2.3 Typing requirements

To use the `ClassDynamic` constructor, which calls `typeOf` to examine type representations, we must declare all type class variables in the context of `Data.Typeable`. We can do this without affecting existing code[2].

[1] As of GHC version 6.8.2, there is a bug in the GHC API pretty-printer that sometimes inserts gratuitous newlines. We must remove these newlines before the resulting code can compile. I anticipate that this bug will be fixed at some point.

[2] Since versions of GHC higher than 6.8 allow type class derivation to be declared after the initial type declaration, we could define instances of `Typeable` for all types for which we need it.

Chapter 5

Conclusions

In man-machine symbiosis, it is man who must adjust: The machines can't. -Alan Perlis

Our investigation of Haskell commenced with the search for a favorable compromise between static correctness and dynamic flexibility for method dispatch. Currently, Haskell's type classes provide a nice framework for grouping types by functionality, but we have no convenient mechanism for using them when types are not known until run time. In this thesis I have shown that we can extend the Haskell framework to support dynamic dispatch of type class methods.

The `ClassDynamic` library I present provides a way to specify dynamic dispatch: the function `classDispatch` essentially allows us to dynamically discriminate membership in type classes for the purpose of method dispatch. I also provide the `dispatchOn` mechanism as a solution to issues of ambiguity in dispatching methods with variable result type. The `dispatchOn` mechanism is useful when the result type of a method dispatch is not known at compile time. By providing respite from the exacting Haskell type checker, dynamic dispatch allows for much more flexibility.

These extensions make dynamic dispatch in Haskell as safe as possible: dynamic method dispatches raise exceptions only in cases when analogous static dispatches would also raise exceptions. For example, the call to

```
read "True" ::  Int
```

raises the run-time exception "no parse" from both static and dynamic dispatch. The new mechanisms use the `Maybe` monad to encapsulate all other forms of error. Using the statically checked monad construct minimizes the points of failure by leveraging the type checker to enforce proper error handling.

In the remainder of this chapter I address future directions for this work and attempt to convince the reader that it is important to continue work in combining dynamic and static type-checking.

5.1 Future work

There are various extensions that would allow for a more seamless integration of the new mechanisms. The most natural extension is to build support for dynamic type class dispatch into the Glasgow Haskell Compiler by having the compiler generate the dictionary and `dispatchOn` functions. This might permit cleaner syntax. For instance, we might be able to shift responsibility of handling the method dictionary to the compiler.

One limitation of the new mechanism is that it does not support separate compilation: the source-to-source translation requires access to the complete source of code, as all instance declarations are relevant. We could modify the Glasgow Haskell Compiler to store type class declaration and instance definition information with compiled binaries. This would allow us to compile files separately and examine the binaries for the necessary information.

We could also think about modifying the language to create more convenient syntax that reflects the grouping of types by type class:

```
printPickledValue :: Dynamic -> IO ()
printPickledValue v =
  case fromDynamicClass Picklable v of
    Maybe (p :: (Picklable a) => a) -> putStrLn p
    Nothing -> putStrLn "Unable to perform request."
```

Another issue that still exists is in the handling of exceptional cases: if a method receives a value of an inappropriate type, the sender of the value has no direct role in the error handling. For instance, consider the following function:

```
printPickledValue :: ClassDynamic -> IO ()
printPickledValue v =
  case classDispatch funDict "pickle" v >>= fromClassDynamic of
    Just s -> putStrLn s
    Nothing -> putStrLn "Can't do anything; we were given a bad value."
```

Since `printPickledValue` does not return a value to its caller, it has no convenient way of communicating an error to the sender of some bad value. It would be useful to devise a language mechanism for throwing an error to the sender.

One of my longer-term goals is to use Haskell concepts in building a better language that simultaneously accommodates prototyping and development for production. Haskell's type system requires programs to be coherent at all times: all values must have a type at compile time and all types must be consistent. When prototyping, we often do not know where we are going, so having such an exacting type system could hinder development. For production code, however, we want the security of static guarantees. I would like to have a language that allows me to create a dynamically

type-checked prototype that can subsequently be annotated for compile-time correctness-checking and optimization. It would be even better to find a way to allow code to coexist in these two phases such that I could 1) run prototyping code with production code and to 2) run flexible dynamic code with code for which static correctness is more critical.

5.2 Looking ahead

Finding a favorable compromise between dynamic flexibility and static correctness will continue to be important. Consider a perhaps unexpectedly prevalent use of the Internet for exchanging computation and data: as of January 2008, the Berkeley Open Infrastructure for Network Computing (BOINC), which manages scientific computing efforts, had 1,237,468 active users downloading computing projects to run on their personal computers [fNC08]. (Compare this to Jabber, a popular open-source platform for chat software, which had 301,434 users at around the same time [Fou08].) BOINC provides evidence that users are willing to accept complex computations from across the network provided that there are some safety guarantees. Developing languages with flexible dynamic features that simultaneously provide technical guarantees on program behavior will liberate people from trust-based networks and allow for more expressive interactions across the network.

5.3 Acknowledgments

Professor Greg Morrisett and Paul Govereau have supplied the crucial inspiration and direction for this project. Despite their busy schedules they always made time for our weekly meetings, without which I would have been much more lost and much less enlightened.

I should also thank the Harvard faculty for an excellent and enjoyable education. Professor Margo Seltzer's good advising is the reason I continued to study computer science. Professor Norman Ramsey's programming languages course showed me the beauty of λ. Professor Radhika Nagpal's enthusiastic guidance helped me to discover my research passions. Professor Stuart Shieber's freshman seminar introduced me to rigorous thinking and writing. Professor Harry Lewis, through a semester's worth of early-morning CS 121 staff meetings, gave me perspective on the purpose of my education.

My fellow students have also been an important part of my college, and therefore thesis-writing, experience. Jesse Tov, whom I first met as my teaching fellow when I was a freshman, has answered my questions through the years as I progressed from Scheme to ML to Haskell. Jie Tang, Thomas Carriero, and Seth Flaxman have given me valuable thesis draft feedback. My friendships with Aliza Aufrichtig, Giselle Barcia, Dara Blume, Brigit Helgen, and Marianne Kaletzky have made me less of a robot and more of a human. Adam Kapor's support and good conversation have increased the quality of this document and my overall quality of life.

Finally, I owe much to my family, who supports me in "all that makes sense."

Appendix

.1 ClassDynamic interface

The public interface of the `ClassDynamic` library is as follows.

- `data ClassDynamic = ClassDynamic [ClassDynamicInst]`

  The type constructors for the `ClassDynamic` type. Rather than storing a single type representation an object, we are storing a list of possible instances. `ClassDynamic` is an instance of `Typeable` and `Show`. Data type `ClassDynamicInst` currently has the same definition as `TypeRep`, but we hope to extend this to accomodate polymorphic type representations.

- `data PolyDictEntry = CDE { name :: String, fns :: [ClassDynamic]}`

  Stores data type for mapping function names to instantiated functions.

- `toclassDyn :: Typeable a => a -> ClassDynamic`

  Wraps an object with a `ClassDynamic` constructor and stores the value with the type representation. Uses `Typeable` 's `typeOf` function to determine the type representation.

- `fromClassDyn :: Typeable a => ClassDynamic -> a -> a`
  `fromClassDynamic :: Typeable a => ClassDynamic -> Maybe a`
  `fromClassDynamicAll :: Typeable a, MonadPlus m => ClassDynamic -> m a`

  Returns the actual representation of the value if it is monomorphic and the request type represents the type representation. Function `fromclassDyn` calls `fromClassDynamic` and returns the second argument (a default value of type `a`) if `fromClassDynamic` returns `Nothing`. Function `fromClassDynamicAll` returns all values of the given type.

- `combineClassDynamic :: MonadPlus m => m ClassDynamic -> Maybe ClassDynamic`

  Takes a list of `ClassDynamic` values and combines them into a single multi-type value if all values are similarly kinded.

- `classDynApply :: ClassDynamic -> ClassDynamic -> Maybe ClassDynamic`
  `classDynApp :: ClassDynamic -> ClassDynamic -> ClassDynamic`

Applies all type/value combinations to all type/value combinations of the argument and creates a new ClassDynamic value containing all new possible

- `classDynTypeReps :: ClassDynamic -> [TypeRep]`

  Returns the lists of types represented by a `ClassDynamic` value.

- `classDispatch :: Map.Map String [ClassDynamic] -> String -> ClassDynamic -> Maybe ClassDynamic`

  Dynamic type class method dispatch function.

.2 Using the ClassDynamic library

Suppose we had the following corresponding dictionary to the previous definition of type class `C`:

```
funDict :: [ClassDictEntry]
funDict =
  [ PDE "c" [toClassDyn (c :: Int -> Bool -> String),
             toClassDyn (c :: Bool -> Bool -> String),
             toClassDyn (c :: Int -> Int -> String)] ]
```

We could use this dictionary to dispatch on the function `c`:

```
main = do {
    let fCDyn = case classDispatch functionMap "c" (toClassDyn (1 :: Int)) of
  Just f -> f
  Nothing -> error "bad"
  ; putStrLn (show fCDyn)
    -- Fully apply it.
  ; let fCAppDyn = classDynApp fCDyn (toClassDyn True)
  ; putStrLn (show fCAppDyn)
    -- Get the result back out.
  ; case fromClassDynamic fCAppDyn of
      Just (s :: String) -> putStrLn s
      Nothing -> putStrLn "no result"
}
```

This code would produce the following output:

```
<<[<<Bool -> Char]>>, <<Int -> [Char]>>]>>
<<[<<[Char]>>]>>
(Int, Bool)
```

Note that the `ClassDynamic` class also worked for types that contain type applications of higher-kinded types. We would declare a type class with such a class as follows:

```
class CCons c a where cCons :: c a -> String
instance CCons [] Int where cCons x = "list"
```

With this, a call to `cCons` with a valid integer list will yield the resulting string `"list"`.

Bibliography

[BS02] 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.

[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[Com08a] Haskell Community. Haskell hierarchical libraries. `http://www.haskell.org/ghc/docs/latest/html/libraries/`, 2008.

[Com08b] Haskell Community. Haskell wiki. `http://www.haskell.org/haskellwiki`, 2008.

[fNC08] Berkeley Open Infrastructure for Network Computing. BOINC combined statistics. `http://boinc.netsoft-online.com/`, 2008.

[Fou08] Jabber Software Foundation. Jabber.org. `http://www.jabber.org/`, 2008.

[HM94] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. Technical report, Pittsburgh, PA, USA, 1994.

[LP03] 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).

[P+03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[SSP97] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. Technical Report TR-1997-26, University of Glasgow, Department of Computing Science, Aug 1997.

[Wei00] Stephanie Weirich. Type-safe cast: Functional pearl. pages 58–67, September 2000.