Dynamic Typing in Haskell

John Peterson

Research Report YALEU/DCS/RR-1022

Yale University

Department of Computer Science

New Haven, CT 06520

D DDTDDDDDTDDD **D**

DDsDDacD

cDDe Dsing DDe DemplaDes pDDDiDeD in DDe Haskell DepDDDD Dynamic Dyping is alsD essenDial DD DDe DeDDgging sysDem Deing DeDelDpeD DDD DDe Dale sysDemD DinallyD Dynamic Dyping can De DseD DD implemenD eDisDenDial DypingD

DDeDiDDs DDDk Dn Dynamic Dyping DDD DD Das DDcDseD Dn DDe Dasic pDDDlem DD mDDing DalDes inDD anD DDD DD DDe Dynamic DDmain in a Dype saDe manneDD DDile DDe sysDem pDesenDeD DeDe Das DDese same capaDiliDiesD De Dill DDcDs Dn DDis issDe DD sDDDcDDDal pDlymDDpDismD DDile sDDDcDDDal pDlymDDpDism is cDmmDnly DseD in Dynamic langDagesD iD is nDD pDesenD in pDeDiDDs DDDk in Dynamic Dyping in pDlymDDpDic langDagesD TDe gDal DD DDis DDDk is DD cDeaDe a simple sysDem DD Dynamic Dypes DDicD is DDDD Dell inDegDaDeD inDD DDe Haskell Dype sysDem anD sDDcienDly eDpDessiDe DD accDmDDaDe sDDDcDDDal pDlymDDpDismD

D TDe Haskell Dlass DysDem

HaskellDDD DeaDDDes an eDDensiDn DD DDe DD Dype sysDem knDDn as *typeclasses' Type classes implement overloading 'ad-hoc polymorphism'' An overloaded function has multiple definitions' one for each type for 'hich the overloading is de'ned' In 'as'ell' a class is an operation or set of operations 'hose 'ehavior is selected 'y the type of o''ect it is applied to' An instance provides a de'nition of the operations in a class for a 'xed type' A class may have many di'erent associated instances' Type classes may 'e associated 'ith type varia'les using a context' A context associates type varia'les 'ith classes and appears in type signatures as a set of class and type varia'le pairs preceeding* `=>`. The signature `Num a => a -> a -> a` would describe a function with two arguments and returned value of the same type, which must be in class `Num`.

The class system is essentially a dispatching mechanism. A call to an operation in a class is forwarded to the handler attached to the argument type by an instance declaration. Type inference ensures that such a handler will always exist.

Type classes are implemented using *dictionaries' 'ach instance declaration de'nes a dictionary' The context of a type signature 'ill 'e manifested at execution time 'y the passing of dictionary parameters' A function such as*

```
f :: Num a => a -> a -> a
```

will be passed a dictionary which contains the implementation for operations in `Num` for whatever type of value is passed to `f`. Dictionary conversion is a process in which hidden dictionary parameters are added to functions during type checking. The implementation of type classes is described in ["].

A "xed set of structurally polymorphic operations in "askell is supplied using a mechanism called *derived instances' Derived instances are code templates de'ned 'y the language standard 'hich must 'e 'ired into the compiler' 'ach use of a derived instance in a type declaration is expanded into type-speci'c source code'*

D Dynamic Typing

Dynamic Dyping Dses a single DypeD DeDe calleD Dynamic, to hold objects of arbitrary type. To ensure type correctness, these values are coupled with a type tag which captures the same sort of type information used during compile time type checking.

The toDynamic construct creates a value of type Dynamic. This pairs a value with the principal type of that value as inferred at compile time. Since it captures a value of any type and returns a dynamic, it resembles a function with the typing

```
toDynamic :: a -> Dynamic
```

Polymorphic typing introduces some complications. The type computed for an object at compile time may contain either universally or existentially quantified type variables. Universal quantification (usually) describes polymorphic functional objects. Consider the following example:

```
id :: a -> a
id x = x
f :: Dynamic
f = toDynamic id
```

Here the type tag in the dynamic value associated with f contains a universally quantified type variable.

When part of an object's type is bound in an outer type environment, existential quantification may be required to describe the type of the object.

```
f :: a -> Dynamic
f x = toDynamic x
```

In this example, the dynamic value captures a parameter of arbitrary type passed to f. In the previous example the type of the id function is completely known at place toDynamic is called. Here, x is bound to an object of a type unknown at compile time. The type tag within the created dynamic thus cannot make use of a universally quantified type variable as the dynamic value holds a specific concrete type.

The issue of existential type quantification has been dealt with in a number of ways. In ["], capture of existentially quantified types is forbidden. This is extended by Leroy and Mauny [2] to allow existential types to be represented in the tags of dynamic values. However, the added expressiveness they achieve seems to be of little practical value. We will avoid this issue at present" for now we will assume that type reconstruction of polymorphic objects is possible. This eliminates the need for existential quantification in type tags. Ways to either perform or avoid type reconstruction are presented in section ". In the above example a type for the parameter x would somehow be computed at run time and inserted into the dynamic. This problem does not arise in dynamicly typed languages since the value of x would include type information.

The inverse of toDynamic is fromDynamic. This unwraps a dynamic value and ensures that the type tag of the dynamic is compatible with (no less general than) the type inferred

DDDing sDaDic Dype analysisD Dike toDynamic, fromDynamic is not actually a function but rather a special construct. It has the following type behavior:

```
fromDynamic :: Dynamic -> a
```

The context in which the fromDynamic appears determines the permissible type tags in the dynamic value. The type of the dynamic must be at least as general as the type inferred at compile time. When this is not the case, a runtime type error occurs. For example, in

```
x :: Int
x = fromDynamic y
```

the type in the dynamic y must indicate that it contains an Int.

A more conventional way of unwrapping dynamic values uses pattern matching. We augment standard Haskell patterns with pat :: type to match dynamic values with the given type. One important restriction on dynamic pattern matching is required: type variables within the signatures cannot propagate outside of the corresponding clause. The following generates a compile time type error since a type variable unwrapped by the dynamic escapes:

```
f :: Text a => Dynamic -> a
f (x :: Text a => a) = x
```

The following would be allowed:

```
f :: Dynamic -> String
f (x :: Text a => a) = show x  -- show :: Text a => a -> String
```

Here, the type variable created by pattern matching against the Dynamic does not escape f and the restriction is not violated.

This restriction is due to the fact that this sort of pattern matching creates existentially quantified types. Such types are only valid locally and cannot escape.

The fromDynamic construct does not have this restriction. Calls to fromDynamic may appear in a context containing existentially quantified type variables, but a runtime error will result unless the runtime instantiation of the existentially quantified type variables exactly matches the type of the Dynamic value. In the function

```
addOneDynamic :: Num a => a -> Dynamic -> a
addOneDynamic x d = x + fromDynamic d
```

the dynamic d must be contain the same type as the parameter x. Since in Haskell addition is defined in conjunction with the class system, it has the typing Num a => a -> a -> a. This function adds numerics of any sort as long as the dynamic is the same numeric type as the parameter x.

The fromDynamic construct cannot be used in situations where the compile time type is more general than the runtime type expected. In the following

```
bad :: Dynamic -> Dynamic -> Dynamic
bad x y = toDynamic (fromDynamic x + fromDynamic y)
```

the calls to fromDynamic appear in a context with the typing Num a => a. The types in the dynamics must be at least as general, so no practical type is allowable for x and y. This problem can be avoided using pattern matching:

```
good :: Dynamic -> Dynamic -> Dynamic
good (x :: Num a => a) y = toDynamic (x + fromDynamic y)
```

Here the value in x is opened up using a pattern match which creates an existential type in the context where fromDynamic is used. The type within y is then compared to the type taken from x as expected. The returned dynamic will have this same type.

The semantics chosen for fromDynamic are in contrast with other approaches to dynamic typing. While previous work with dynamic typing provides toDynamic and dynamic pattern matching, the use of fromDynamic to allow dynamic values to re-integrate in existentially typed situations is novel. Although the difference between the semantics of pattern matching and fromDynamic in unwrapping dynamic values may not seem obvious, each is necessary in different situations.

The Static Type System at Runtime

The tag created by the toDynamic function encodes compile time type information. For this information to be useful at run time, the compile time type environment must be present during execution. While the compile time type environment is usually discarded before runtime in languages with static type systems, it is commonly part of the runtime environment in dynamic languages such as Lisp.

The type environment is represented at runtime by a set of constants which correspond to the various type declarations. In Haskell, the following abstract types are used to represent the type environment: DataType, Constructor, Class, and Instance. The type of a Dynamic can be extracted by the dType function. It returns a value of type Signature, defined as follows:

```
type Context = [Class]
data Signature = Signature [Context] Type
data Type = Tycon DataType [Type] |
            Tyvar Int
```

The Signature constructor introduces numbered, universally quantified type variables with class constraints, one for each element in the list of contexts. The definitions of the DataType, Constructor, Class, and Instance are not shown here: these are complex types whose internal structure is accessed through a set of helper functions. There are too many such functions to list here" we will introduce them only as needed. A naming convention is used in which all such function begin with d. The user is not allowed to create new data types, classes, or instances at runtime.

These functions retrieve the type of a dynamic value:

```
dType :: Dynamic -> Signature  -- Get the type of a dynamic
dConstructor :: Dynamic -> Constructor  -- Get the constructor associated
                          -- with the data value in a dynamic
```

The dType function merely returns the tag in a dynamic value. It does not force evaluation of the object captured by the dynamic. In contrast, dConstructor evaluates the captured value and returns the data constructor associated with the value.

While the availability of type information during execution is traditional in dynamic languages, the the usefulness of this type information has not been generally recognized.

"   Manipulating Dynamic Values

While toDynamic and fromDynamic allow conversion into and out of Dynamic, the ability of a program to perform operations within the dynamic domain is of great importance. These operations allow dynamic objects to be safely manipulated from outside the context of their local type system.

For an algebraic sum of products data type, two dynamic operations are available: construction and selection. The selection operation destructures a dynamic value into a set of dynamic components, each carrying the appropriate type. Selection is accomplished by dSlots:

```
dSlots :: Dynamic -> [Dynamic]
```

Construction creates a new dynamic value. Each value of type Constructor has an associated construction function. This function is retrieved by the dMake function:

```
dMake :: Constructor -> [Dynamic] -> Dynamic
```

The dMake function must perform limited type inference to construct a correctly typed result. Type errors may occur: these are manifested as a special data type encoded as a Dynamic:

```
data DynamicError = DynamicError String
```

The string contains an error message from the type checker.

The dConstructor, dSlots, and dMake functions allow values of arbitrary type to be examined or created in a dynamic context.

Another basic operation on dynamic values is the dynamic function call. This applies a function (encoded as a Dynamic) to a list of dynamic arguments. Function calling also performs type inference: the function and argument types must agree and a result type must be determined. The function

```
dynamicApply :: Dynamic -> [Dynamic] -> Dynamic
```

performs function application in the dynamic domain. It also runs the type checker to compute the result type of the application. The result of dynamic application is a value of type DynamicError whenever any type mismatch occurs.

Ds Haskell is a pDDely DDncDiDnal langDageD DDncDiDn calling DD impDDe DDncDiDns mDsD De DanDleD a DiD DiDeDenDlyD TDe DDncDiDn

```
dynamicApplyIO :: Dynamic -> [Dynamic] -> IO Dynamic
```

calls functions in the IO monad. The IO monad is used by Haskell to sequence operations on the global state" the function called by dynamicApplyIO must also be in the IO monad.

" Type Reconstruction

Reconstructing the type of a polymorphic object at runtime is a problem that has been studied in the context of memory management and debuggers. The simplest approach is to use runtime tags even in a staticly typed language. Another is to infer types on demand by searching through the runtime stack for the point at which a polymorphic type is instantiated["]. In Haskell, the class system supplies a simple means of recovering runtime type information. The class

```
class Typable a where
  typeof :: a -> Signature
```

can be attached to existentially quantified type variables enabling the recovery of the associated type at runtime. All types are placed into this class implicitly and the appropriate typeof function derived by the compiler. The class mechanism will then propagate the correct type signature where it is needed without further action by the user. While this looks suspiciously like a retreat to fully tagged data objects, there is a significant difference. Tags are not physically attached to the values" the class system will maintain the tag as a separate structure. One tag value may apply to many data values. For example, in

```
f :: Typable a => [a] -> [a] -> [a]
```

a single tag value passed to f would identify the polymorphic components of all list structures used by f. In contrast, full dynamic typing would require a tag attached to every element of each list.

The use of typable has a few drawbacks, however. The Typable context may appear unexpectedly when dynamic typing is used. While correct context propagation will occur without user intervention, user supplied type signatures would need to mention Typable in some cases. It is possible to hide Typable from the user by implicitly propagating this class in for all polymorphic values. This complicates function calling somewhat and may slow down execution. A more serious drawback is that abstraction boundaries are broken: the inner structure of all types are passed around at runtime is available to the user through conversion to the Dynamic type.

A less intrusive approach is to create invent new types on the fly to denote the type of unknown objects. This is known as Skolemization in logic and the newly generated types are Skolem constants. These Skolem types may be associated with type classes in the same manner as ordinary types. In the function

```
f :: Text a => a -> Dynamic
f x = toDynamic x
```

the Skolem type generated to denote the type of x will be a member of the class Text. The dictionary passed to f is captured in the new Datatype object created by the call to toDynamic. A dynamic created by f can be used in any situation requiring a value in the class Text. While a Skolem type does not match the actual type of the value it is assigned to, it serves both as a container to hold class assertions which accompany the data value and as a marker that allows objects created with the same skolem type to be recognized later as sharing a common type. The following function captures two numerics:

```
capture x y :: Num a => a -> a -> (Dynamic,Dynamic)
capture x y = (toDynamic x,toDynamic y)
```

Since x and y are declared to have a common type by the type signature of capture, only one skolem type is generated and this type is saved in the tag of both dynamics. This allows these dynamics to be used in a context where two dynamics of the same type are necessary, such as the good function presented earlier:

```
good :: Dynamic -> Dynamic -> Dynamic
good (x :: Num a => a) y = toDynamic (x + fromDynamic y)
```

When Skolem types are used, type signatures play an especially important role. Without the attached type signature, the capture function would not know that the two dynamics share a common type or that the type is in class Num.

An advantage to Skolemization is that it acts as an abstraction barrier, giving a local name to an unknown type.

Skolem types must be propagated through pattern matching against dynamics. The existential types introduced by pattern matching are found at runtime in the type tag in the dynamic object. In the good function, the type a is instantiated by pattern matching against the first argument. The type of x thus corresponds to the tag of the dynamic which binds x. When toDynamic is called, the value of this tag can then be used in the new dynamic value, preserving the type.

While Skolemization is used during type inference in [2], these types are not used in the tags of dynamics as we do. The utility of Skolem types derives from the class system - without type classes, they are much less useful.

Unfortunately, Skolemization results in a loss of referential transparency. If the capture function is called repeatedly with the same arguments it will create a new Skolem type in each pair of dynamics. To avoid this problem, dynamic objects containing Skolem types should not have their types matched against skolem types created in different contexts.

" Implementation Issues

At compile time, the primary implementation issues are type capture and the insertion of runtime unification. The expression toDynamic exp is compiled as

```
let temp = exp in MkDynamic (captureType temp) temp
```

where MkDynamic is the internal data constructor for dynamics. The captureType instructs the compiler to create a runtime type value from the type of temp.

The expansion of fromDynamic is similar. The runtime uni"cation function takes two types: a "more general' and a "less general' and returns either an error indication or a list of bindings of type variables in the "more general' pattern. The expression fromDynamic exp is replaced by

```
temp where
 MkDynamic tag value = exp
 temp = case unify tag (captureType temp) of
          UnifyError -> error "Dynamic type error"
          _          -> value
```

"attern matching is expanded in a manner similar to fromDynamic except that the bindings created during uni"cation may be required to instantiate dictionaries. Uni"cation errors yield failure to match instead of a runtime error. Support to locate dictionaries at runtime is required. Including dictionary conversion, the code generated by the type checker for

```
f (x :: Text a => a) = show x
```

would be

```
f x' = case unify <constant for Text a => a> tag of
        UnifyError -> failure
        Match [a_type] ->
          case dictLookup a_type "Text" of
            NotFound -> failure
            Found dText -> show dText x
  where
    MkDynamic tag x = x'
    failure = error "Runtime pattern match failure"
```

The implementation of type capture can be accomplished in a manner similar to dictionary conversion[ ]. Type constants cannot be completely constructed until all components of the type have been generali"ed. "laceholders are used to postpone type capture on type variables still bound in the type environmen a very simple macro-like mechanism invoked by deriving clause which will call a dynamic version of the instance function. The declaration

```
derive Text(T) as
instance Text(Components T) => T Type where
  print x = printDynamic (toDynamic x)
```

The only non-standard "askell we use here is Components Type. This is expanded into a list of all structure components of the type. The declaration

```
data Type a = C1 a | C2 Int deriving Text
```

would generate the following instance:

```
instance Text(a,Int) => Text (Type a) where
  print x = printDynamic (toDynamic x)
```

DDnDeDD DeDDcDiDn is DeDDiDeD DD cDDDecDly pDDpagaDe cDnDeDD inDDDmaDiDn DDDn DD DDe cDmD pDnenDs DD DDe DypeD Ds DDis is nDD alDays pDssiDleD DDe insDance may nDD De DeDiDaDleD TDis cDnDeDD DeDDcDiDn is alDeaDy a paDD DD DDe DeDiDeD insDance eDpansiDn D DDe Dnly DiDeDence is DDaD iD DecDmes mDDe DisiDle DeDeD TDis `derive ... as` construct is not directly related to the dynamic typing" alternative ways of generating this instance declaration could be proposed. "hat is crucial here is that the body of the `Text` instance de"ned above calls a generic printer capable of printing values of any type. Although we have used a very simple syntactic means to start the dynamic printing process, the real work is being done by dynamic types.

The dynamic printer is responsible for printing values of arbitrary type encoded as dynamics. The type captured by `toDynamic` will be guaranteed to have a `Text` instance for each component of the type.

A very simple printer for dynamic values would be:

```
printDynamic :: Dynamic -> String
printDynamic d = typeName ++ concat slotValues d where
  typeName = constrName (dConstructor d)
  slotValues = map (\(s :: Text a => a) -> "(" ++ print s ++ ")") dSlots d
```

Although this is much simpler than the standard "askell print instance it illustrates how dynamics can be used to create a structurally polymorphic print routine. The context supplied in the instance declaration assures that the `(s :: Text a => a)` will always match the value in the slot of the dynamic.

The `Eq` (equality) class illustrates some more aspects of dynamic typing. "ere the deriving template is

```
derive Eq(T) as
instance Eq(Components T) => Eq T where
  x == y = dymanicEq (toDynamic x) (toDynamic y)
```

The dynamic equality function "rst ensures that the values and created with the same constructor and then compares slots pairwise.

```
dynamicEq :: Dynamic -> Dynamic -> Bool
dynamicEq x y = dConstrName x == dConstrName y &&
                   sameSlots (dSlots x) (dSlots y)
  where sameSlots [] [] = True
        sameSlots ((x :: Eq a => a):xs) (y:ys) =
            x == fromDynamic y && sameSlots xs ys
```

The function `dConstrName` returns the name of the constructor.

The dynamic reader di"ers from the previous examples in that it is type driven rather than data driven. The read process takes a type ob"ect and returns an ob"ect of that type. The template for the read function is

```
derive Read(T) as
```

```
 instance Read(Subtypes T) => Read T where
  read s = a where
    ty = toDynamic a
    a = (fromDynamic v,s) where
         (v,s) = (readDynamic s ty)
```

This read function returns a tuple containing the value and the remaining input. The class system supplies the type being read: the typing of read is

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

The value ty in the above example is a dynamic constructed strictly for the purpose of extracting the type tag. The value a has the type returned by the read function so ty will contain the desired type. Since a is the result of the readDynamic function, an attempt in the function to evaluate ty would lead to an infinite loop. Since Haskell is lazy, however, the dynamic can be constructed without evaluating the value it contains. (For a strict language such as ML a version of toDynamic which captures only the type and not the value could be used.)

The readDynamic function is too involved to present here. At the heart of this function is dMake, used to create the value, and a form of dSlots, which returns the slot types of the object being read. These types are used to recursively read the structure components.

"   Summary and Conclusions

Much of what is presented here is derived from either ML style dynamic typing [2, "] or general programming practice in dynamic languages. The principal contributions are the use of the more complex Haskell type system, the skolemization process to avoid type reconstruction, and the semantics of the fromDynamic operator. This work has practical value: it provides a elegant solution to a serious shortcoming in the design of Haskell. By providing a user an accessible means to achieve structural polymorphism, a previously inflexible aspect of the language has been made accessible to the user.

Other typing systems with capabilities similar to