Dynamic Typing in Haskell

John Peterson

Research Report YALEU/DCS/RR-1022

Yale University

Department of Computer Science

New Haven, CT 06520

Abstract

Static type systems, as used in such languages as C, ML, and Haskell, perform all type inference at compile time; during execution the type of an object is implicit. Dynamic typing, as used in Lisp or Smalltalk, requires the type of an object to be explicitly encoded as part of its value. Each of these approaches has advantages: static typing detects type errors at compile time and has no runtime overhead. Dynamic typing is more expressive. In this paper we study the incorporation of dynamic types within the framework of the Haskell type system, a derivative of the Hindley - Milner system used in ML. Unlike ML, the Haskell type system allows overloading (ad-hoc polymorphism) using type classes.

The system of dynamic typing described here builds on work done in the ML community. In addition to coercion to and from the domain of dynamic types, we allow for the direct manipulation of objects within the dynamic domain in a type safe manner. Of particular practical importance is the ability to generalize an operation over arbitrary types based on the structure of the data type. This dynamic typing system has been implemented as a part of the Yale Haskell system and dynamic types have been used to build the 'derived instance' component of the Haskell language in a much more general manner than presently used.

1 Introduction

Static typing, as represented by the ML type system, has many well known advantages: a very expressive type language, complete type error detection at compile time, and run time efficiency. In Haskell, the expressiveness of this type system is increased to support ad-hoc polymorphism (overloading). Dynamic typing, as found in the Lisp family of languages, also has significant advantages. Using dynamic typing, a function or program can interact with types not within the compile time type environment. Another very important aspect of dynamic typing in the Lisp world is the ability to access the description of a type. This type description (or meta-type) of an object allows for operation on completely arbitrary types. For example, a printer can use the type of an unknown object to find out how to access the components of the object for printing.

We define a function as *structurally polymorphic* when its behavior is governed by the structure of the type of object it is applied to. Unlike parametric polymorphism, this behavior is not identical for all types. Structural polymorphism differs from the ad-hoc polymorphism supplied by the Haskell type system in that a single definition of the function serves for all types, while ad-hoc polymorphism simply allows a different definitions to be associated with different types. Many common operations are structurally polymorphic, including structural equality, generalized mapping and folding, and printing.

Previous work on dynamic typing [2, 3] has focused on the basic problem of moving values into and out of the dynamic domain in a type safe manner. While the system presented here has these same capabilities, we will focus on this issue of structural polymorphism. While structural polymorphism is commonly used in dynamic languages, it is not present in previous work in dynamic typing in polymorphic languages. The goal of this work is to create a simple system of dynamic types which is both well integrated into the Haskell type system and sufficiently expressive to accomodate structural polymorphism.

The dynamic typing system decribed here has been implemented as a part of the Yale Haskell system.

2 The Haskell Class System

Haskell[1] features an extension to the ML type system known as *typeclasses*. Type classes implement overloading (ad-hoc polymorphism). An overloaded function has multiple definitions, one for each type for which the overloading is defined. In Haskell, a *class* is an operation or set of operations whose behavior is selected by the type of object it is applied to. An *instance* provides a definition of the operations in a class for a fixed type. A class may have many different associated instances. Type classes may be associated with type variables using a *context*. A context associates type variables with classes and appears in type signatures as a set of class and type variable 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*. Each instance declaration defines a dictionary. The context of a type signature will be manifested at execution time by 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 [4].

A fixed set of structurally polymorphic operations in Haskell is supplied using a mechanism called *derived instances*. Derived instances are code templates defined by the language standard which must be wired into the compiler. Each use of a derived instance in a type declaration is expanded into type-specific source code.

3 Dynamic Typing

Dynamic typing uses a single type, here 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 [3], 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 6. 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 during static type analysis. Like `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.

4 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.

5 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.

As Haskell is a purely functional language, function calling to impure functions must be handled a bit differently. The function

```
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.

6 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[5]. 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.

7 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 unification 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
```

Pattern matching is expanded in a manner similar to `fromDynamic` except that the bindings created during unification may be required to instantiate dictionaries. Unification 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[4]. Type constants cannot be completely constructed until all components of the type have been generalized. Placeholders are used to postpone type capture on type variables still bound in the type environment. As outer expressions are generalized, these type placeholders may be resolved in one of three ways:

- The type may be instantiated to a concrete type. This replaces the placeholder with a type constant.
- The type may be associated with a data object. The type of the data object must either by computed using the `Typable` type class or a skolem type can be generated.
- The type may be instantiated through dynamic pattern matching. In this case, a type object is available at runtime and is placed into the type object.

This example captures the type of a parameter:

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

The type `a` is captured in both dynamics. This would be transformed into something like:

```
f x y numDictionary = (MkDynamic atype x, MkDynamic atype y) where
    atype = makeSkolemType [("Num",numDictionary)]
```

This uses the skolemization; the `typeOf` style would be similar. In either case, the two dynamic types created will have identical type tags and can thus be used in a common `Num` context.

Other changes to the compiler are relatively straightforward. Type declarations must generate type constants. Separate compilation requires that the linkage between types and classes be postponed until load time. These constants contain functionals which implement the `dConstructor`, `dSlots`, and `dMake` functions. The type checker must check for escaping type variables in dynamic pattern matching.

The runtime support needed by dynamic typing consists mainly of the type unification algorithm made available at runtime. This is not significantly different from the unification done at type checking time in the compiler.

8 An Example: Derived Instances

To demonstrate the use of dynamics to achieve structural polymorphism, we will present an implementation of some of the functions currently defined as derived instances in Haskell. We combine the textual (macro expansion) approach with dynamics to allow the majority of the derived instance to be placed in a dynamic handler. A small, type specific bit of code in the actual instance is used to convert values to dynamic form and call the handler.

Derived instances are created by `deriving` clause in a data declaration. Instead of building source code for the entire instance, we propose 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 Haskell 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)
```

Context reduction is required to correctly propagate context information down to the components of the type. As this is not always possible, the instance may not be derivable. This context reduction is already a part of the derived instance expansion - the only difference is that it becomes more visible here. This `derive ... as` construct is not directly related to the dynamic typing; alternative ways of generating this instance declaration could be proposed. What is crucial here is that the body of the `Text` instance defined 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 Haskell 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. Here 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 first 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 differs from the previous examples in that it is type driven rather than data driven. The read process takes a type object and returns an object 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.

9 Summary and Conclusions

Much of what is presented here is derived from either ML style dynamic typing [2, 3] 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 dynamic typing have been proposed, including soft typing[7] and quasi-static typing[6] address slightly different issues. Both increase the expressiveness of the type system in an implicit manner, while the dynamic typing here must be fully explicit. Neither addresses the issue of structural polymorphism. However, both view dynamic types as having all other types as a subset. This allows for implicit conversion to the dynamic domain, something not found in this approach. Such implicit conversion is actually quite valuable since it would eliminate the need for `toDynamic` and `fromDynamic`. However, such implicit conversion involves serious alterations to the basic type system. Our approach leaves the Haskell type system itself untouched.

10 Acknowledgments

This work was supported by grants from DARPA, contract number N00014-91-J-4043, and from NSF, contract number CCR-9104987.

References

[1] P. Hudak, S.L. Peyton Jones and P. Wadler (eds.). Report on the programming language Haskell, version 1.2. *ACM SIGPLAN notices*, 27, 5, May 1992.

[2] X. Leroy and M. Mauny. Dynamics in ML. In *Functional Programming Languages and Computer Architecture*, Cambridge, MA, pages 406-426, Springer, Aug. 1991. Lecture Notes in Computer Science, Vol 523.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. In *Transactions on Programming Languages and Systems*, 13, 2, April 1991.

[4] J. Peterson and M. Jones. Implementing Type Classes. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 227-236, June 1993.

[5] B. Goldberg Tag-Free Garbage Collection for Strongly Typed Programming Languages. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165-176, June 1991.

[6] S. Thatte. Quasi-static Typing. In *Conference Record of the Seventeenth Annual Symposium on Principles of Programming Languages*, pages 367-381, January 1990.

[7] R. Cartwright and M. Fagan. Soft Typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278-292, June 1991.