Implementing Type Classes

John Peterson and Mark Jones
Department of Computer Science, Yale University,
P.O. Box 2158 Yale Station, New Haven, CT 06520-2158, USA.
Electronic mail: peterson-john@cs.yale.edu, jones-mark@cs.yale.edu.

Abstract

We describe the implementation of a type checker for the functional programming language Haskell that supports the use of *type classes*. This extends the type system of ML to support overloading (ad-hoc polymorphism) and can be used to implement features such as equality types and numeric overloading in a simple and general way.

The theory of type classes is well understood, but the practical issues involved in the implementation of such systems have not received a great deal of attention. In addition to the basic type checking algorithm, an implementation of type classes also requires some form of program transformation. In all current Haskell compilers this takes the form of dictionary conversion, using functions as hidden parameters to overloaded values. We present efficient techniques for type checking and dictionary conversion. A number of optimizations and extensions to the basic type class system are also described.

1 Introduction

In the study of programming languages, the term *overloading* is used to describe the ability of a single symbol to have different interpretations as determined by the context in which it appears. Standard examples of this include the use of **+** to represent both addition of integers and addition of floating point quantities, or the use of **==** to compare both character values and pointers. In each case, the intended meaning of the overloaded symbol can be determined from the types of the arguments to which it is applied.

One common approach is to completely resolve overloading at compile time. The compiler installs type specific meanings for all overloaded symbols, based either on type information attached to operands (the usual case) or some more general overloading resolution mechanism. A significant drawback to this approach is that overloaded operations cannot be abstracted while retaining their overloaded nature.

A more dynamic approach to overloading which preserves the ability to abstract overloaded definitions is found in object oriented languages. Here, resolution of overloaded operations occurs at run time. There are two particular problems to be dealt with:

- How do we determine which interpretation of an overloaded operator should be used in any particular situation? There are many examples for which the appropriate overloading cannot be determined at compile time. For example, in a program that uses the function $double = \lambda x.x + x$ to double both integer and floating point values, there is no way to fix any single interpretation for the $+$ symbol.

- How do we ensure that overloaded values are only ever used with appropriate arguments? For example, it would probably not make sense to try to add two character values. As a result, we must also ensure that the *double* function is never applied to a character value.

Standard ML uses two different approaches to overloading:

- The type of each arithmetic operator such as **+** must be uniquely determined from its context, possibly by inserting an explicit type declaration. This compile time resolution of overloaded operators is not able to preserve the full overloading of **+** in the double function; one specific implementation of **+** must be chosen.

- Standard ML introduces a notion of equality types to deal with the typing of the equality function. This is undesirable because it forces the programmer to accept a particular *structural* definition of equality – one which tests for equality of representation rather than equality of represented value. In addition, Appel [1] reports that "Equality types add significant complexity to the language and its implementation".

An alternative approach to the treatment of overloading was introduced by Wadler and Blott [11] based on the notion of a *type class* and is intended to provide a uniform and general framework for solving exactly these kinds of problems. Type classes are most widely known for their use in the functional programming language Haskell [6] where they are used mostly to deal with standard primitive functions such as **+** and **==**. In addition, we have also found that type classes can be useful in more specific application areas where they can help to produce clear and modular programs [7]. We should also mention that there does not appear to be any significant reason why the use of type classes should be limited to non-strict, purely functional languages: in principle, any language based on the basic Hindley/Milner/Damas type system could be extended to support the use of type classes.

This paper is meant to serve as a practical guide for the implementation of type classes. Previous work in this area has concentrated on the typing rules and has culminated in a set of syntax-directed typing derivations which are the basis for our type checker. Here we will use the typing rules to create a concrete algorithm which both type checks and transforms the program. We hope to reveal the essential simplicity of both the theory and implementation of type classes. Our concerns are type checking programs efficiently, generating the best possible code from the type checker, and introducing a number of simple extensions to type classes which can be incorporated into our basic framework. This work is the result of our experience implementing type classes in both the Yale Haskell compiler and the Gofer interpreter.

2 Type classes – a summary

We begin by summarizing the main features of a system of type classes for a very simple and well known example – the definition of an equality operator, written as `==`, that is:

- **polymorphic**: use of the operator is not restricted to values of any single type.
- **overloaded**: the interpretation of equality is determined by the types of its arguments.
- **extensible**: the definition of equality can be extended to include new datatypes.

Our example programs will be written using the concrete syntax of Haskell with explanatory comments where necessary. Further details may be found in [6]. We will also use the following terms while describing the class system:

method A primitive overloaded operator such as `==` will be called a method. Methods are found in expressions.

class A group of related methods is packaged into a class. Each class has a name which is used in the type language.

data type Type classes use the same sort of data types used by the ML type system. A type constructor names a data type in the type language while data constructors create values in the expression language.

instance An instance binds a data type to operations which implement the methods of a specified class for that type.

The basic idea is to define a set of types `Eq`, known as a type class in Haskell, that contains precisely those types for which a suitable definition of equality has been given using an instance declaration. The definition of the class `Eq` is as follows:

```
class Eq a where
    (==)  ::  a -> a -> Bool
```

The first line introduces a name for the class and indicates that the type variable `a` will be used to represent an arbitrary instance of the class in the following part of the definition. (In the general case, we use an expression of the form `C t` to represent the assertion that the type `t` is an instance of the class `C`.) The remaining part of the declaration lists a collection of method functions which are associated with the class. In this particular example, there is only a single method function, written as an infix operator, `==`. The type signature `a -> a -> Bool` indicates that, for each instance `a` of `Eq`, the `==` symbol behaves like a function that takes two arguments of type `a` and returns a value of type `Bool`.

A class declaration may also define a set of superclasses for a given class. The use of superclasses does not significantly complicate this type system and will be discussed later.

Using the notation of Haskell, the full type of `==` is written as `(Eq a) => a -> a -> Bool`. Note the convention that all free variables in a type expression are implicitly bound by a universal quantifier at the outermost level. Thus `==` is 'polymorphic' in `a`, but the choice of types for `a` is restricted to instances of `Eq`. Type class constraints like this are often described as the *context* part of a type.

Even before we have defined a single instance of the class, we can use the `==` operator, either indirectly or directly, to define other values. The restriction to instances of `Eq` is reflected in the types assigned to these values. For example:

```
member          :: Eq a => a -> [a] -> Bool
member x []      = False
member x (y:ys) = x==y || member x ys
```

The first line of this definition gives the type of `member`. Note that, in Haskell, `[a]` represents the type of lists of values of type `a`. As in the basic ML type system, user supplied type signatures are not actually required since they can be inferred automatically by the type system. We provide such signatures in our examples as documentation. The second and third lines are typical of the way that functions are defined in Haskell. In this example there are two equations, using pattern matching on the left hand side to distinguish between the two cases when the list argument to `member` is empty, `[]`, or non-empty, written `(y:ys)` where `y` and `ys` are the head and tail, respectively, of the list.

The types which are members of a class are defined by a collection of *instance declarations* which may be distributed throughout the program, typically in different program modules where new datatypes are introduced. For built-in types, the definition of equality may well be provided by a primitive function:

```
instance Eq Int where
    (==)  =  primEqInt
```

More generally, we can define instances of the class `Eq` for any built-in and user-defined algebraic data types as in the following definition of equality on lists:

```
instance Eq a => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x==y && xs==ys
    _      == _      = False
```

(The underscore character `_` in the last line is used as a wildcard; it indicates that, if neither of the first two cases can be applied, the the equality test will produce a result of `False`.) The expression `Eq a => Eq [a]` in the first line indicates that the definition of equality on lists depends on the definition of equality used for the elements held in the list: if `a` is an instance of `Eq`, then so is `[a]`.

The set of types defined by a finite collection of instance declarations may be infinite (but recursively enumerable). For example, the definitions given above describe the equality operator for integers, lists of integers, lists of lists of integers and so forth.

3 Implementing Overloading

One standard technique used in the implementation of run-time overloading is to attach some kind of *tag* to the concrete representation of each object. Overloaded functions such as the equality operator described above can be implemented by inspecting the tags of their arguments and dispatching the appropriate function based on the tag value. Many schemes exist for the encoding of tags to make the tag dispatch efficient. This is essentially the method used to deal with the equality function in Standard ML of New Jersey [2]. One of the benefits of static type checking is that it provides a compile-time check which ensures that the the equality function will never be applied to an object for which there is no corresponding definition of equality.

Unfortunately, the use of tags as described above has a number of drawbacks. It can complicate data representation and may not be well suited to the underlying hardware. Perhaps more significantly, there are some forms of overloading that cannot be implemented using this approach. In particular, it is not possible to implement functions where the overloading is defined by the returned type. A simple example of this is the `read` function used in Haskell to parse a string as a value of any type that is an instance of the `Text` class, the set of readable (and printable) types.

An elegant way to avoid these problems is to separate objects from their tags, treating tags as data objects in their own right. For example, we can implement `read` as a function that takes an extra argument which gives the tag of the result value. This amounts to passing type information around at run-time but this is only necessary when overloaded functions are actually involved. This is potentially more efficient than uniformly tagging every data object regardless how it will be used.

Using this approach, the `member` function in the previous section might be implemented by translating the original definition to:

```
member' :: (a -> a -> Bool) -> a -> [a] -> Bool
member' eq x []     = False
member' eq x (y:ys) = eq x y || member' eq x ys
```

In other words, the implementation of `member` is simply parameterized by the appropriate definition of equality. The tag in this case is the equality function itself.

In this example, we could evaluate `member 2 [1,2,3]` by rewriting it as `member' primEqInt 2 [1,2,3]` and evaluating that expression instead. For a more interesting example, if `xs` is a list of lists of integers, then we could evaluate `member [1] xs` in a similar way, rewriting it as `member' (eqList primEqInt) [1] xs`, where:

```
eqList :: (a -> a -> Bool) -> [a] -> [a] -> Bool
eqList eq []     []     = True
eqList eq (x:xs) (y:ys) = eq x y &&
                          eqList eq xs ys
eqList eq _      _      = False
```

The definition of `eqList` can be obtained directly from the instance declaration on lists in Section 2 in much the same way as the definition of `member'` was obtained from that of `member`. Type classes do not require a particular definition of equality for a data type; any function of the appropriate type can by supplied by the user to check equality. As a convenience, Haskell allows the programmer to use *derived instances* for some of the standard classes like `Eq`, automatically generating appropriate instance definitions. Note that this feature is not itself part of the underlying type system.

One of our goals in the remainder of this paper is to describe how these *translations* can be obtained automatically as part of the type checking process.

4 Static Analysis

Before type checking, the compiler must assemble the components of the static type environment. The data type, class, and instance declarations (all top-level declarations in Haskell) must be collected and processed. One constraint on these declarations is that instances must be unique: only one instance declaration for a particular combination of data type and class is allowed. This ensures that the meaning of overloaded operations with respect to parameter data types is consistent throughout the program.

In the previous section we described how the `member` function can be implemented by parameterizing its definition with respect to an implementation of the `==` method. In the general case, a class may have several different methods and it is sensible to parameterize the definitions of overloaded functions using *dictionary* values; tuples containing implementations for each of the methods for a particular instance of a class.

Static analysis generates a dictionary for each instance declaration and these dictionaries themselves may be overloaded. When a dictionary contains overloaded functions, as manifested in the context component of an instance declaration, it will reference further subdictionaries when constructed. A dictionary containing `eqList` would need to be overloaded to provide the `eq` argument to `eqList`. In our implementation, this captured dictionary is stored by partially applying `eqList` to just the `eq` argument when the dictionary containing `eqList` is created.

Each instance can be converted to a 4-tuple containing the data type, the class, a dictionary, and the context associated with the instance. A definition is inserted into the program which binds the dictionary value, a tuple of method functions, to a variable, the dictionary variable. The instance context can be represented by a list of class constraints, one class constraint for each argument to the data type defined by the instance. A class constraint is the (possibly empty) list of classes which must apply to the constituent type.

The instance declaration for list equality would create this dictionary:

```
d-Eq-List = eqList
```

and the declaration itself would be represented by:

```
(List,Eq,d-Eq-List,[[Eq]])
```

where `List` is the name of the list type data constructor. Since this class has only one method a tuple is not needed;

normally a dictionary would be tuple containing a definition for each method. The context indicates that the first argument to the `List` type constructor must be in the `Eq` class.

Dispatching a method requires selection of the appropriate function from a dictionary. *Selector* functions which retreive a method from a dictionary are also defined as the static type environment is processed. (In the previous example no selectors are needed since there is no tuple in the dictionary.) These simply extract a component of a dictionary tuple, a constant time operation since each member function is located at a specific place in the dictionary. Dictionaries are only used where overloading cannot be resolved at compile time. When the type associated with a method is known at compile type the type specific version of the method is called directly without using the dictionary.

5 Type Inference

We will separate the issues of type inference, in which each program expression is assigned a (possibly overloaded) type, and dictionary conversion, in which the program code is transformed to explicitly extract method functions from dictionaries.

The use and implementation of ML style type inference is well documented and we will not repeat this here (see [4] for example). Instead, we concentrate on the relatively minor changes that are needed to extend ML style type inference with support for type classes.

As in ordinary ML typechecking, type variables and unification play a central role. Type variables are initially unbound, corresponding to 'unknown' types. As type checking proceeds, various constraints on the values that can be assigned to type variables are exposed, for example by ensuring that the argument type of a given function is the same as the type of the value to which it is actually applied. These constraints are solved by instantiating unbound type variables to more accurate types. Type classes require an additional field in each uninstantiated type variable: the context, a set (represented by a list) of classes.

Unification is affected in a very simple way: when a type variable is instantiated, its class constraints must be passed on to the instantiated value. If this is another type variable, its context is augmented, using set union, by the context of the instantiated variable. When a context is passed on to a type constructor *context reduction* is required. Context reduction uses the instance declarations in the static type environment to propagate all class constraints to type variables.

The type constructor being reduced by context reduction must be an instance of the reducing class. If not, type checking fails with an error that an attempt has been made to use an overloaded operator at a type that is not an instance of the corresponding class. If an instance declaration is found linking the data type and the class, the context of the instance declaration propagates to the type constructor arguments. This process continues until contexts have been propagated exclusively to type variables.

As an example, consider the unification of `Eq a => a`, a type variable with an `Eq` context, and the type `[Integer]`. The type variable is instantiated to `[Integer]`. Before context reduction, the resulting type is `Eq [Integer] => [Integer]`. The instance declaration for class `Eq` over the list data type exists (otherwise a type error occurs) and propagates the context `Eq` to the argument to the list type constructor. This leads to the type `Eq Integer => [Integer]`. Now we can see that the program must also include an instance declaration that makes `Integer` an instance of the class `Eq`. Assuming that this is true, and since the `Integer` type constructor does not take any arguments, no further constraints can exist leaving only `[Integer]` as the resulting type. Note, however, that the unification would have failed if the required instance declarations were not found in the static type environment. By a similar process, unification of `Eq a => a` and `[b]` would yield the type `Eq b => [b]`. Here, contexts remain attached to the resulting type variables.

The following code implements type variable instantiation in the presence of type classes. Each type variable has a value field which is either null (uninstantiated) or contains an instaniated type. The context field is a list of classes attached to uninstantiated type variables. The `findInstanceContext` function searches the static type environment for an instance with the selected class and data type. If not is found this function signals a type error. It returns a list of contexts, one for each argument to the data type.

```
instantiateTyvar(tyvar,type)
 tyvar.value := type
 propagateClasses(tyvar.context,type)

propagateClasses(classes,type)
 if tyvar(type)
  then type.context := union(classes,type.context)
  else for each c in classes
        propagateClassTycon(c,type)

propagateClassTycon(class,type)
 s = findInstanceContext(type.tycon,class)
 for each classSet in s, typeArg in tycon.args
  propagateClasses(classSet,typeArg)
```

One other minor change to ML type inference is required. When a `letrec` is typechecked all variables defined by the letrec share a common context. This will be discussed in Section 8.3.

It is worth emphasizing that context reduction is the only significant change to the ML type inference process necessary to infer correct typings for Haskell programs involving type classes. On the other hand, dictionary conversion, as described in the following section (or some similar process), must be carried out to implement overloading in the final executable version of the type checked program.

6 Dictionary Conversion

Dictionary conversion affects the generated code in two ways. First, overloaded definitions receive additional parameter variables to bind dictionaries. Second, a reference to an overloaded definition must be passed dictionaries. Thus the typechecker needs only two basic changes: when reference to an overloaded definition (which is usually a function but may be of any type) is type checked the hidden dictionary parameters must be inserted. When a definition (either at the top level or in a local definition using a `let` or `letrec`) is typed hidden dictionary arguments are inserted to bind any necessary dictionaries needed to resolve the overloading at run time.

The relation between a type signature and dictionary parameters is simple: each element of the context corresponds to a dictionary passed into or recieved by an overloaded definition. For example, a function with the type `(Eq a, Text b) => a -> b` would require two dictionaries, one for the class `Eq` and another for `Text`. The ordering of a context is arbitrary; dictionaries can be passed in any order so long as the same ordering is used consistently.

Adding dictionary passing code to the program during the code walk performed by the standard ML typechecker is perhaps the essential implementation issue addressed here. The type associated with an expression may change due to unification as the type checker proceeds. Since types only stablize at generalization the appropriate dictionaries needed to resolve overloading cannot be determined until the entire expression being generalized has already been walked over. To avoid a second pass over the code after generalization, we will hold onto the necessary bits of unresolvable code using *placeholders*. A placeholder captures a type and an object to be resolved based on that type. During generalization, placeholders are replaced by the required type-dependent code.

6.1 Inserting Placeholders

Placeholders are inserted when the type checker encounters either an overloaded variable, a method, or a letrec bound variable. Slightly different forms of placeholder are used in each case.

Overloaded variables are rewritten as an application to placeholders that will ultimately be replaced by the dictionaries implied by the variable's context. The fresh type variables associated with the variable are captured in the placeholders. For example, if `f` has type `(Num a, Text b) => a -> b`, the type checker will first freshly instantiate the type variables in `f`, yielding a typing of `(Num t1,Text t2) => t1 -> t2`. This fresh instantiation of type variables is part of ordinary ML style type checking. The value `f` will be rewritten as an application: `f <Num,t1> <Text,t2>`. The <*object*,*type*> notation will be used to represent placeholders. These placeholders become additional arguments to `f` which will be placed ahead of any other arguments. The classes `Text` and `Num` which appear in the placeholders indicate that the placeholder must resolve to an expression yielding a dictionary for that class.

Method functions are converted directly to placeholders. The type variable in the placeholder corresponds to the type variable which defines the class in the class declaration. For example, the `==` method in class `Eq` would be typechecked by freshly instantiating its type, yielding `Eq t1 => t1 -> t1 -> Bool`, and returning the placeholder `< == ,t1>`. Since the object in the placeholder is a method, it will be resolved to either a specific implementation of the method (if the type variable becomes instantiated to a concrete type) or code to select a `==` function from an `Eq` dictionary.

Recursively defined variables cannot be converted until their type is known. References to such variables encountered before they are generalized are simply replaced by a placeholder until the correct context has been determined. For example, in a simple recursive definition such as `member`, the recursive call to `member` becomes a placeholder until its type is generalized. After generalization, it is treated as an ordinary overloaded variable.

6.2 Inserting Dictionary Parameters

Once a definition has been typed, any context associated with the type variables in the definition is used to generate dictionary parameter variables which will bind the dictionaries needed to resolve the overloading. This occurs during the generalization portion of type inference. Generalization gathers all uninstantiated type variables in the type of a definition and creates a new dictionary variable for every element of every context in these type variables. A lambda which binds the dictionaries is wrapped around the body of the definition and a parameter environment is created. This environment is used to resolve placeholders created during typechecking of the definition. This environment maps a pair containing a class and type variable onto a dictionary parameter variable.

As a simple example, if the inferred type of `f` is `(Num t1,Text t2) => t1 -> t2`, then the definition of `f` is changed to `f = \d1 d2 -> f'` where `f'` is the original definition of `f`. This creates the following parameter environment: `[((Num,t1),d1),((Text,t2),d2)]`.

6.3 Resolving Placeholders

At generalization, placeholders inserted into a definition can be resolved. A list of all placeholders, updated as each new placeholder is created, can be used to avoid walking through the code in search of placeholders. After dictionary parameters have been inserted, each placeholder is examined. For placeholders associated with either methods or classes, the type associated with the placeholder determines how it will be resolved. There are four possibilities:

1. The type is a type variable in the parameter environment. In this case, the mapping defines a variable which will carry the dictionary at run-time. A class placeholder is resolved to the dictionary parameter variable; a method placeholder requires a selector function to be applied to the dictionary variable.

2. The type has been instantiated to a type constructor. An instance declaration associated with this type supplies either the method itself for a method placeholder or a dictionary variable for a class placeholder. Since dictionaries or methods themselves may be overloaded the type checker may need to recursively generate placeholders to resolve this additional overloading.

3. The type variable may still be bound in an outer type environment. The processing of the placeholder must be deferred to the outer declaration.

4. If none of the above conditions hold, an ambiguity has been detected. The ambiguity may be resolved by some language specific mechanism or simply signal a type error.

Placeholders associated with recursive calls can be resolved in two different ways. The simplest way is to generate an overloaded variable reference which is no different than for other overloaded variables. This can only be done after generalization since the context of the recursive call is unknown until this time. However, since any dictionaries passed to a recursive call remain unchanged from the original entry to the function, the need to pass dictionaries to inner recursive calls can be eliminated by using an inner entry point where the dictionaries have already been bound. An example of this is shown in section 7.

7 Examples

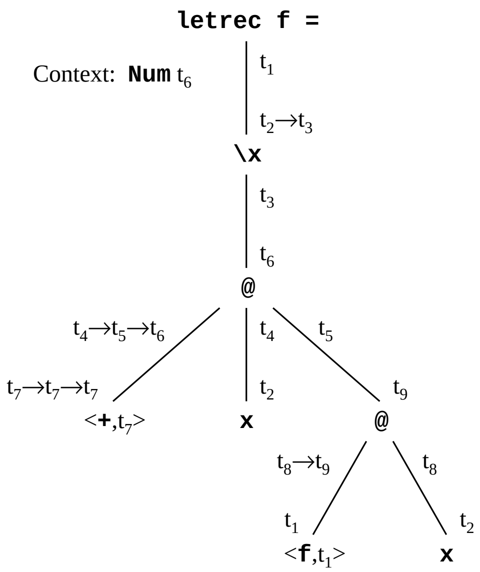
We will illustrate the operation of our type checker with a couple of examples, each of which consists of three code trees. The first code tree shows freshly instantiated type variables (the t_i) and inserted placeholders. The rules for instantiating type variables and the type templates are the same as for ML type checking. The second tree shows the result of unification. Types are unified pairwise along the lines in the diagrams. Finally, the result of generalization and placeholder resolution will be shown. The actual type checker performs unification continuously instead of after all type variables have been instatuated; these steps are separated here for clarity.

Rather than write the context associated with a type variable each time it is mentioned all type variable context information will be shown at the side.

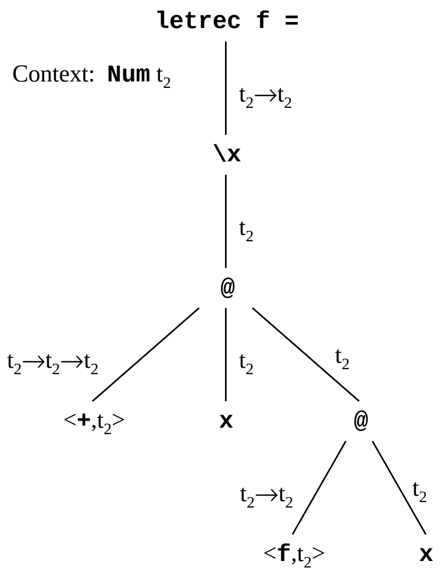
The following function `f` uses a method, `+`, and a recursive call to itself.

```
class Num a where
    (+) :: a -> a -> a

f = \x -> x + f x
```

Type variable instantiation and placeholder insertion produce the following expression tree. The @ nodes are curried applications.

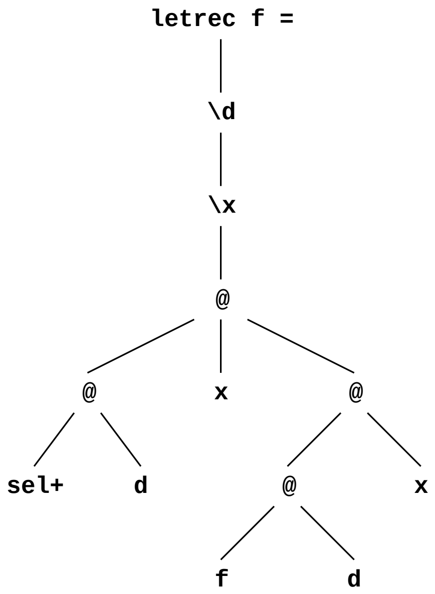


After unification, this becomes:



The type in the placeholder associated with `+` is part of the parameter environment. This indicates that a dictionary passed into `f` will contain the implementation of `+` appropriate for the parameter `x`. At execution time, the `sel+` function will retrieve this addition function from the dictionary.

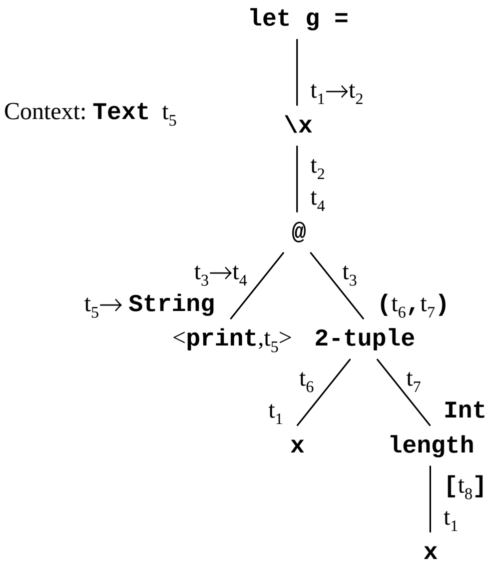
This is the simplest translation in which the recursive call passes the dictionary `d` unchanged. A better choice would have been to create an inner entry to `f` after `d` is bound and use this for the recursive call to avoid passing `d` repeatedly.



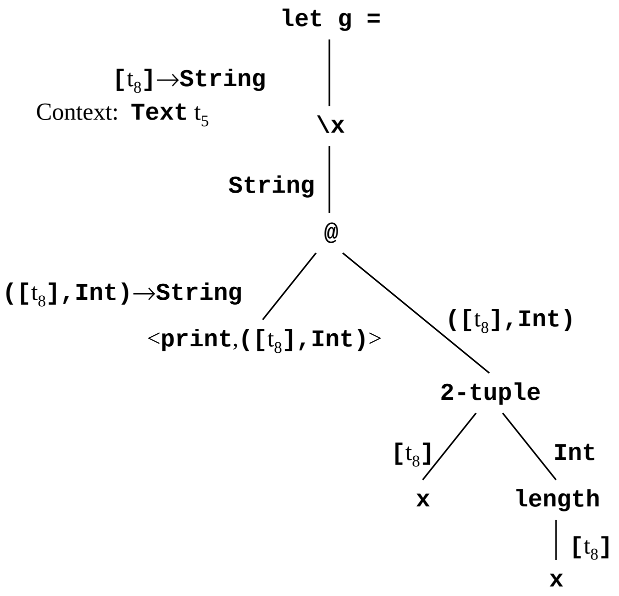
The next example uses a previously defined overloaded function, `length`, with type `[a] -> Int`. The necessary class and instance declarations are included. We will use the convention that dictionaries are named d-*class*-*type*.

```
class Text a where print :: a -> String

instance (Text a, Text b) => Text (a,b) where
    print = print-tuple2

instance Text Int where ....
instance Text a => Text [a] where ....

g = \x -> print (x,length x)
```

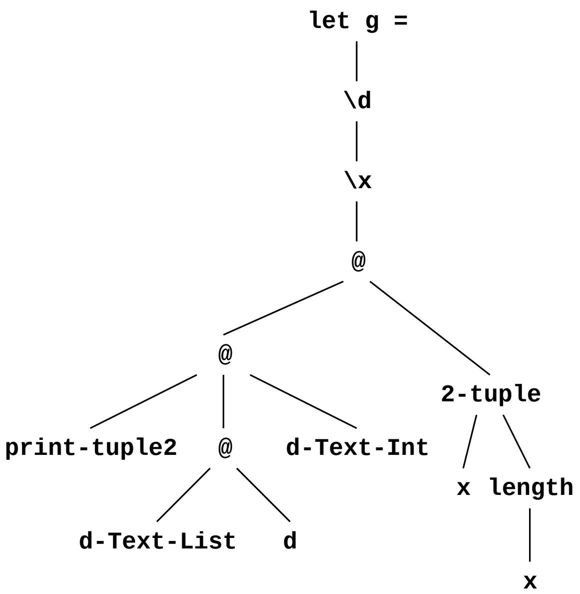
After placeholder insertion and type variable instantiation:



After unification, this becomes:



The placeholder is resolved to a specific printer for 2-tuples. As this function is overloaded, further placeholder resolution is required for the types associated with the tuple components.



8 Extensions

This implementation of type classes can be extended in a number of ways to both improve the generated code and increase the expressiveness of the type system.

8.1 Using a Class Hierarchy

In a Haskell class declaration, a set of classes may be declared as superclasses of the defined class. For example, in the declaration:

```
class Text a => Num a where
    ...
```

class `Text` is a superclass of `Num`. This implies that all datatypes declared to be in class `Num` must also be declared to be in `Text`. This superclass relation allows a type such as `(Num a, Text a) => a` to be abbreviated as `Num a => a`.

Within the type checker, superclasses require few changes. When class sets for type variables are constructed, contexts implied by the superclass relation can be removed. This compacts the class sets and requires fewer dictionary parameters. Superclasses also require that dictionaries contain all superclass dictionaries. During dictionary conversion, a dictionary may not be directly available if the associated class has been absorbed as a superclass. In this case, the dictionary or method must be fetched from an embedded superclass dictionary. Dictionary representation affects the speed of method selection. Deeply nested dictionaries can be avoided by flattening dictionaries to include all methods in both the associated class and in all superclasses at the top level of the structure. This slows down dictionary construction but speeds up selection operations. The effect of this tradeoff in real programs is not yet known. Optimizations which avoid dictionary construction make flattening more attractive.

8.2 Default Method Declarations

Class declarations may supply a default method to be used when an instance declaration does not provide an implementation of a method in the class. This requires only that this definition bound to a variable for use during dictionary construction. This variable is placed into any dictionary in which the method is not specified by the instance declaration.

8.3 Typing Recursive Definitions

So far we have assumed that the letrec construct binds only one variable. Mutually recursive definitions can be understood as a tupling of functions. Mutually recursive functions `f` and `g` could be defined as follows:

```
letrec (f,g) = (fbody,gbody) in ...
```

Here there is only a single recursive value, the tuple. Notice that the context of `f` and `g` are combined by this translation. Although mutually recursive functions are not actually implemented as tuples, they are type checked in this manner.

All functions defined by a single letrec share a common context. This may create ambiguous functions when the type of a letrec bound variable does not contain the full context of the letrec. Such a function can be called within the letrec but not from outside. This is not an error in itself but the compiler provides a warning about such functions.

While it is easy for a single recursive function to use a local entry point to avoid passing dictionaries to recursive calls, this is harder to do for more than one function. It is simplest to pass all dictionaries to each recursive call within the letrec. Otherwise, all outside entries into the recursive group of functions needs to be funneled through a single lambda binding all dictionaries and then some sort of switch is required to enter the proper function. Other approaches may be possible but this does not seem to be a critical performance issue.

8.4 Reducing Constant Dictionaries

Another source of inefficiency are local functions which are inferred to have an overloaded type but are used at only one overloading. These can be detected during optimization or during type inference. During type inference, this involves saving the type variables created by freshly instantiation of the signature as it is referenced. If all of these variables are instantiated to the same concrete type the dictionary can be reduced to a constant. Flow analysis of the dictionaries can accomplish this same task and is perhaps superior since optimizations may remove some function calls which would prevent a dictionary from being marked invariant.

8.5 Overloaded Methods

Haskell allows method functions to be overloaded in more than the type variable defined by class. For example, a class definition may contain:

```
class Foo a where
    m1 :: Bar b => a -> b
    m2 :: a -> a
```

Here, `m1` contains an extra overloading. A dictionary for this class should have a type `(Bar b => T -> b,T -> T)` for some type `T` in the class `Foo`. That is, the first component should be an overloaded function with `Bar` in the context while the second component is independent of `Bar`. Unfortunately, this type signature is not valid since the context will float outside the tuple. In implementation terms, the tuple will attempt to bind a dictionary for `Bar` when the dictionary is constructed instead of simply placing a function which binds a `Bar` dictionary inside the tuple. This requires the implementation of such dictionaries to go outside the standard type class system when generating such dictionaries. The cleanest solution to this problem would probably involve existential types. The Yale compiler avoids this issue using an internal construct similar to a type cast.

8.6 User Supplied Signatures

User supplied type signatures are a very necessary part of the type system. They can be used to avoid unwanted overloading and are essential for efficiency. Unlike the ML type system, user supplied signatures have a significant impact on the generated code, possibly replacing higher order function calls (method selectors) with direct calls to instance functions.

While there are numerous ways of implementing these signatures, our system does this in a very clean way using read-only type variables. Type variables in signatures are marked as read-only to prevent type instantiation from violating the signatures. A read-only type variable cannot be instantiated or have its context augmented.

Another use of user-supplied signatures is to fix the ordering of dictionaries during dictionary conversion. Haskell uses interface files to support separate compilation. These interfaces provide the signature of each definition in a module. These interface signatures define a specific ordering on the dictionaries passed to resolve overloading; at the implementation level the types `(Foo a,Bar b) => a -> b` and `(Bar b,Foo a) => a -> b` are different in a very important way. The compiler must be aware of any interface for the module being compiled and use that signature to determine the dictionary ordering during generalization.

8.7 The Monomorphism Restriction

The Haskell report [6] imposes a constraint known as the *monomorphism restriction* on the generalization of overloaded variables. This is intended to avoid problems with the loss of laziness that can occur when an overloaded variable is translated to a function with one or more dictionary parameters. Explicit type signatures can be used to avoid the monomorphism restriction in those cases where overloading would otherwise be restricted. Regardless of how the monomorphism issue is treated, it has a very simple implementation. When this restriction applies to a variable, type variables in its context must not be generalized: they must remain in the type environment to avoid fresh instantiation while the body of the defining `let` expression is type checked.

8.8 Avoiding Unnecessary Dictionary Construction

Overloaded dictionaries are not constants and will be constructed dynamicly at run-time. The algorithm presented here may repeatedly reconstruct identical copies of overloaded dictionaries if the underlying implementation is not fully-lazy.

To illustrate how this problem can occur, consider the following implementation of the equality on lists in essentially the same form given by [11]:

```
eqList d []     []     = True
eqList d (x:xs) (y:ys) = eq d x y &&
                         eq (eqDList d) xs ys
eqList d _      _      = False
```

The `eqDList` function constructs a dictionary for equality on lists of type `[a]` given a dictionary `d` for equality on values of type `a`. The `eq` function denotes the selector which extracts the method for `==` from a corresponding dictionary. As it is written, many implementations of this definition will repeat the construction of the dictionary `eqDList d` at each step of the recursion. One simple way to avoid this is to rewrite the definition in the form:

```
eqList d
 = let eql = eq (eqDList d)
       eqa = eq d
       e []     []     = True
       e (x:xs) (y:ys) = eqa x y && eql xs ys
       e _      _      = False
   in e
```

As a further example of the same thing, consider a function `doOne` of type `C a -> a -> Bool` for some class `C` and suppose that the definition of this function requires the construction of a dictionary value. Note that this fact may well be hidden from the compilation system if the definition of `doOne` appears in an external module.

Now suppose that we define a function:

```
doList []     = []
doList (x:xs) = doOne x : doList xs
```

A naive implementation of `doList` might use the definition:

```
doList d []     = []
doList d (x:xs) = doOne d x : doList d xs
```

Any attempt to evaluate the complete list produced by an application of this function will repeat the construction of the redex `doOne d` (and hence repeat the dictionary construction in `doOne`) for each element in the argument list.

Happily, the same observation also makes the solution to this problem quite obvious; we need to abstract not just the dictionaries involved but also the application of overloaded operators to dictionaries, giving the translation:

```
doList d = doList'
 where doList' []     = []
       doList' (x:xs) = doOne' x : doList' xs
       doOne'         = doOne d
```

An additional benefit of this is that the garbage collector can reclaim the storage used for dictionary values as soon as the implementations of the required methods have been extracted from it.

Note that these problems will not occur in an implementation that supports full laziness. Indeed, in each of the examples above, the improved translation can be obtained from the original version using a translation to fully-lazy form as described in [9].

9 Performance Issues

How do type classes affect the compiler? Our observation is that they increase compilation time only slightly. A minor increase in the cost of unification and the placement and resolution of placeholders make up the majority of the extra processing required for type classes.

As far as program execution is concerned, type classes have two costs: the extra level of indirection when dispatching a method function and the time and space required to propagate dictionaries through overloaded functions. The cost of instance function dispatch is actually quite small since this requires only a reference to a tuple element followed by a function call. For all but the simplest method functions this should be negligible. The cost of dictionary creation and propagation is harder to pin down. Passing and storing extra arguments to overloaded functions will incur slightly more function call overhead. Only overloaded dictionaries consume a non-constant amount of space. However, for code which does not use overloaded functions (but still may use method functions) the class system adds no overhead at all since the specific instance functions are called directly. In the case of a lazy language such as Haskell the overhead of overloaded functions may be greater since overloading is implemented using higher order functions. Higher order functions may be much more expensive in Haskell than ML since it is much harder to apply strictness or uncurrying optimizations. This is very noticable for very simple functions such as basic arithmetic operators but for more complex functions, such as in the I/O system, the overhead of overloading is not noticable.

It is possible to completely eliminate dynamic method dispatch within an overloaded function at specific overloadings by creating type specific clones of overloaded function. This could be implemented in a more general partial evaluation context or be controlled through program annotations.

10 Conclusions and Related Work

While type classes are a relatively new addition to type theory, we argue that they should no longer be considered exotic or experimental. Type classes provide an elegant solution to a number of serious language design problems and should be considered as an important tool in programming language construction. Type classes provide a simple and regular framework by which a program can be parameterized. They do not provide the expressiveness of, for example, the ML module system where the parameterization is explicit. On the other hand, they are particularly convenient for some applications because the code needed to support overloading is handled automatically by the compiler.

We have shown an implementation of type classes which is relatively simple, requiring only a few extensions to the basic ML type checking algorithms. The addition of type classes does not severely impact either compiler or program performance.

The basis for a translation from the Haskell syntax for declaring and using type classes was set out by Wadler and Blott [11] and some results from an early implementation based directly on these ideas have been presented by Hammond and Blott [5]. Further ideas, mostly at a fairly abstract level, were presented in the static semantics for Haskell [10] and also, concentrating on the problems of repeated dictionary construction, in [8]. Some of the techniques used to improve the performance of Haskell overloading in the Chalmers Haskell compiler are described in [3]. In summary, experience with the use and development of Haskell systems has done much to reduce the costs of type class overloading.

11 Acknowledgments

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

References

[1] A.W. Appel. A critique of Standard ML. Princeton University CS-TR-364-92, February 1992.

[2] A.W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[3] L. Augustsson. Implementing Haskell overloading. To appear in *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *8th Annual ACM Symposium on Principles of Programming languages*, 1982.

[5] K. Hammond and S. Blott. Implementing Haskell type classes. *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland. Workshops in computing series, Springer Verlag.

[6] 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.

[7] M.P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, Volume 2, Part 4, October 1992.

[8] M.P. Jones. Qualified types: Theory and Practice. D. Phil. Thesis. Programming Research Group, Oxford University Computing Laboratory. July 1992.

[9] S.L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, **21**(5), May 1991.

[10] S.L. Peyton Jones and P. Wadler. A static semantics for Haskell (draft). Manuscript, Department of Computing Science, University of Glasgow, February 1992.

[11] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Principles of Programming Languages*, 1989.