Type Classes and Functional Dependencies

Mark P Jones
Oregon Graduate Institute
September 10, 1999
(Misc typos fixed: October 6, 1999, with thanks to Stephen Eldridge and Martin Erwig for bringing them to my attention!)

[The new feature described here has been implemented in the September 1999 release of Hugs 98, and the text of this note is included in the user manual for that release.]

Introduction

Multiple parameter type classes seem to have many potentially interesting applications. However, some practical attempts to use them have failed as a result of frustrating ambiguity problems. This occurs because the mechanisms that are used to resolve overloading are not aggressive enough. Or, to put it another way, the type relations that are defined by a collection of class and instance declarations are often too general for practical applications, where programmers might expect stronger dependencies between parameters. In the rest of this section we will describe these problems in more detail. We will also describe the mechanisms introduced in the September 1999 release of Hugs that allow programmers to declare explicit dependencies between parameters, avoiding these difficulties in many cases, and making multiple parameter classes more useful for some important practical applications.

1. Ambiguity problems

During the past ten years, many Haskell users have looked into the possibility of building a library for collection types, using a multiple parameter type class that looks something like the following:

```
class Collects e ce where
    empty  :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool
```

The type variable `e` used here represents the element type, while `ce` is the type of the container itself. Within this framework, we might want to define instances of this class for lists or characteristic functions (both of which can be used to represent collections of any equality type), bit sets (which can be used to represent collections of characters), or hash tables (which can be used to represent any collection whose elements have a hash function). Omitting standard implementation details, this would lead to the following declarations:

```
instance Eq e => Collects e [e] where ...
instance Eq e => Collects e (e -> Bool) where ...
instance Collects Char BitSet where ...
instance (Hashable e, Collects a ce)
           => Collects e (Array Int ce) where ...
```

All this looks quite promising; we have a class and a range of interesting implementations. Unfortunately, there are some serious problems with the class declaration. First, the `empty` function has an ambiguous type:

```
empty :: Collects e ce => ce
```

By ‘ambiguous’ we mean that there is a type variable `e` that appears on the left of the => symbol, but not on the right. The problem with this is that, according to the theoretical foundations of Haskell overloading, we cannot guarantee a well-defined semantics for any term with an ambiguous type. For this reason, Hugs rejects any attempt to define or use such terms:

```
ERROR: Ambiguous type signature in class declaration
*** ambiguous type : Collects a b => b
*** assigned to    : empty
```

We can sidestep this specific problem by removing the `empty` member from the class declaration. However, although the remaining members, `insert` and `member`, do not have ambiguous types, we still run into problems when we try to use them. For example, consider the following two functions:

```
f x y = insert x . insert y
g     = f True 'a'
```

for which Hugs infers the following types:

```
f :: (Collects a c, Collects b c) => a -> b -> c -> c
g :: (Collects Bool c, Collects Char c) => c -> c
```

Notice that the type for `f` allows the two parameters `x` and `y` to be assigned different types, even though it attempts to insert each of the two values, one after the other, into the same collection. If we’re trying to model collections that contain only one type of value, then this is clearly an inaccurate type. Worse still, the definition for `g` is accepted, without causing a type error. As a result, the error in this code will not be flagged at the point where it appears. Instead, it will show up only when we try to use `g`, which might even be in a different module.

2. An attempt to use constructor classes

Faced with the problems described above, some Haskell programmers might be tempted to use something like the following version of the class declaration:

```
class Collects e c where
   empty  :: c e
   insert :: e -> c e -> c e
   member :: e -> c e -> Bool
```

The key difference here is that we abstract over the type constructor `c` that is used to form the collection type `c e`, and not over that collection type itself, represented by `ce` in the original class declaration. This avoids the immediate problems that we mentioned above:

- `empty` has type `Collects e c => c e`, which is not ambiguous.

- The function `f` from the previous section has a more accurate type:

  ```
  f :: (Collects e c) => e -> e -> c e -> c e
  ```

- The function `g` from the previous section is now rejected with a type error as we would hope because the type of `f` does not allow the two arguments to have different types.

This, then, is an example of a multiple parameter class that does actually work quite well in practice, without ambiguity problems.

There is, however, a catch. This version of the `Collects` class is nowhere near as general as the original class seemed to be: only one of the four instances in Section 1 can be used with this version of `Collects` because only one of them---the instance for lists---has a collection type that can be written in the form `c e`, for some type constructor `c`, and element type `e`.

3. Adding dependencies

To get a more useful version of the `Collects` class, Hugs provides a mechanism that allows programmers to specify dependencies between the parameters of a multiple parameter class. [For readers with an interest in theoretical foundations and previous work: The use of dependency information can be seen both as a generalization of the proposal for ‘parametric type classes’ that was put forward by Chen, Hudak, and Odersky, or as a special case of the later framework for *improvement* of qualified types. The underlying ideas are also discussed in a more theoretical and abstract setting in a forthcoming manuscript, where they are identified as one point in a general design space for systems of implicit parameterization.]

To start with an abstract example, consider a declaration such as:

```
class C a b where ...
```

which tells us simply that `C` can be thought of as a binary relation on types (or type constructors, depending on the kinds of `a` and `b`). Extra clauses can be included in the definition of classes to add information about dependencies between parameters, as in the following examples:

```
class D a b | a -> b where ...
class E a b | a -> b, b -> a where ...
```

The notation `a -> b` used here between the `|` and `where` symbols---not to be confused with a function type---indicates that the `a` parameter uniquely determines the `b` parameter, and might be read as "`a` determines `b`." Thus `D` is not just a relation, but actually a (partial) function. Similarly, from the two dependencies that are included in the definition of `E`, we can see that `E` represents a (partial) one-one mapping between types.

More generally, dependencies take the form `x1 ... xn -> y1 ... ym`, where `x1`, ..., `xn`, and `y1`, ..., `yn` are type variables with `n`>0 and `m`>=0, meaning that the `y` parameters are uniquely determined by the `x` parameters. Spaces can be used as separators if more than one variable appears on any single side of a dependency, as in `t -> a b`. Note that a class may be annotated with multiple dependencies using commas as separators, as in the definition of `E` above. Some dependencies that we can write in this notation are redundant, and will be rejected by Hugs because they don’t serve any useful purpose, and may instead indicate an error in the program. Examples of dependencies like this include a -> a, a -> a a, a ->, etc. There can also be some redundancy if multiple dependencies are given, as in a->b, b->c, a->c, and in which some subset implies the remaining dependencies. Examples like this are not treated as errors. Note that dependencies appear only in class declarations, and not in any other part of the language. In particular, the syntax for instance declarations, class constraints, and types is completely unchanged.

By including dependencies in a class declaration, we provide a mechanism for the programmer to specify each multiple parameter class more precisely. The compiler, on the other hand, is responsible for ensuring that the set of instances that are in scope at any given point in the program is consistent with any declared dependencies. For example, the following pair of instance declarations cannot appear together in the same scope because they violate the dependency for D, even though either one on its own would be acceptable:

```
instance D Bool Int where ...
instance D Bool Char where ...
```

Note also that the following declaration is not allowed, even by itself:

```
instance D [a] b where ...
```

The problem here is that this instance would allow one particular choice of [a] to be associated with more than one choice for b, which contradicts the dependency specified in the definition of D. More generally, this means that, in any instance of the form:

```
instance D t s where ...
```

for some particular types t and s, the only variables that can appear in s are the ones that appear in t, and hence, if the type t is known, then s will be uniquely determined.

The benefit of including dependency information is that it allows us to define more general multiple parameter classes, without ambiguity problems, and with the benefit of more accurate types. To illustrate this, we return to the collection class example, and annotate the original definition from Section 1 with a simple dependency:

```
class Collects e ce | ce -> e where
   empty  :: ce
   insert :: e -> ce -> ce
   member :: e -> ce -> Bool
```

The dependency ce -> e here specifies that the type e of elements is uniquely determined by the type of the collection ce. Note that both parameters of Collects are of kind *; there are no constructor classes here. Note too that all of the instances of Collects that we gave in Section 1 can be used together with this new definition.

What about the ambiguity problems that we encountered with the original definition? The empty function still has type Collects e ce => ce, but it is no longer necessary to regard that as an ambiguous type: Although the variable e does not appear on the right of the => symbol, the dependency for class Collects tells us that it is uniquely determined by ce, which *does* appear on the right of the => symbol. Hence the context in which empty is used can still give enough information to determine types for both ce and e, without ambiguity. More generally, we need only regard a type as ambiguous if it contains a variable on the left of the=> that is not uniquely determined (either directly or indirectly) by the variables on the right.

Dependencies also help to produce more accurate types for user defined functions, and hence to provide earlier detection of errors, and less cluttered types for programmers to work with. Recall the previous definition for a function`f`:

```
f x y = insert x y = insert x . insert y
```

for which we originally obtained a type:

```
f :: (Collects a c, Collects b c) => a -> b -> c -> c
```

Given the dependency information that we have for`Collects`, however, we can deduce that `a` and `b` must be equal because they both appear as the second parameter in a`Collects` constraint with the same first parameter`c`. Hence we can infer a shorter and more accurate type for `f`:

```
f :: (Collects a c) => a -> a -> c -> c
```

In a similar way, the earlier definition of`g` will now be flagged as a type error.

Although we have given only a few examples here, it should be clear that the addition of dependency information can help to make multiple parameter classes more useful in practice, avoiding ambiguity problems, and allowing more general sets of instance declarations.