Type Classes with Functional Dependencies*

Mark P. Jones

Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
Beaverton, Oregon, USA
mpj@cse.ogi.edu

Abstract. Type classes in Haskell allow programmers to define functions that can be used on a set of different types, with a potentially different implementation in each case. For example, type classes are used to support equality and numeric types, and for monadic programming. A commonly requested extension to support 'multiple parameters' allows a more general interpretation of classes as relations on types, and has many potentially useful applications. Unfortunately, many of these examples do not work well in practice, leading to ambiguities and inaccuracies in inferred types and delaying the detection of type errors.
This paper illustrates the kind of problems that can occur with multiple parameter type classes, and explains how they can be resolved by allowing programmers to specify explicit dependencies between the parameters. A particular novelty of this paper is the application of ideas from the theory of relational databases to the design of type systems.

1 Introduction

Type classes in Haskell [11] allow programmers to define functions that can be used on a set of different types, with a potentially different implementation in each case. Each class represents a set of types, and is associated with a particular set of *member functions*. For example, the type class *Eq* represents the set of all equality types, which is precisely the set of types on which the (==) operator can be used. Similarly, the type class *Num* represents the set of all numeric types—including *Int*, *Float*, complex and rational numbers—on which standard arithmetic operations like (+) and (−) can be used. These and several other classes are defined in the standard Haskell prelude and libraries [11, 12]. The language also allows programmers to define new classes or to extend existing classes to include new, user-defined datatypes. As such, type classes play an important role in many Haskell programs, both directly through uses of the member functions associated with a particular class, and indirectly in the use of various language constructs including a special syntax for monadic programming (the do-notation).

* The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-96-C-0161.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 230–244, 2000.

The use of type classes is reflected by allowing types to include *predicates*. For example, the type of the equality operator is written:

$$(==) :: Eq\ a \Rightarrow a \to a \to Bool$$

The type variable a used here represents an arbitrary type (bound by an implicit universal quantifier), but the predicate $Eq\ a$ then restricts the possible choices for a to types that are in Eq. More generally, functions in Haskell have types of the form $P \Rightarrow \tau$, where P is some list of predicates and τ is a monotype. If P is empty, then we usually abbreviate $P \Rightarrow \tau$ as τ. In most implementations, the presence of a predicate in a function's type indicates that an implicit parameter will be added to pass some appropriate *evidence* for that predicate at run-time. For example, we might use an implementation of equality on values of type a as evidence for a predicate of the form $Eq\ a$. Details of this implementation scheme may be found elsewhere [14].

In a predicate such as $Eq\ a$, we refer to Eq as the *class name*, and to a as the *class parameter*. Were it not for the use of a restricted character set, constraints like this might instead have been written in the form $a \in Eq$, reflecting an intuition that Eq represents a set of types of which a is expected to be a member. The Haskell syntax, however, which looks more like a curried function application, suggests that it might be possible to allow classes to have more than one parameter. For example, what might a predicate of the form $R\ a\ b$ mean, where two parameters a and b have been provided? The obvious answer is to interpret R as a two-place relation between types, and to read $R\ a\ b$ as the assertion that a and b are related by R. This is a natural generalization of the one parameter case because sets are just one-place relations. More generally, we can interpret an n parameter class by an n-place relation on types.

One potential application for multiple parameter type classes was suggested (but not pursued) by Wadler and Blott in the paper where type classes were first described [14]. The essence of their example was to use a two parameter class *Coerce* to describe a subtyping relation, with an associated coercion operator:

$$coerce :: Coerce\ a\ b \Rightarrow a \to b.$$

In the decade since that paper was published, many other applications for multiple parameter type classes have been discovered [13]; we will see some of these in later sections of the current paper. The technical foundations for multiple parameter classes have also been worked out during that time, and support for multiple parameter type classes is now included in some of the currently available Haskell implementations. So it is perhaps surprising that support for multiple parameter type classes is still not included in the Haskell standard, even in the most recent revision [11]. One explanation for this reticence is that some of the proposed applications have not worked particularly well in practice. These problems often occur because the relations on types that we can specify using simple extensions of Haskell are too general for practical applications. In particular, they fail to capture important dependencies between parameters. More concretely, the use of multiple parameter classes can often result in ambiguities and inaccuracies in inferred types, and in delayed detection of type errors.

In this paper, we show that many of these problems can be avoided by giving programmers an opportunity to specify the desired relations on types more precisely. The key idea is to allow the definitions of type classes to be annotated with functional dependencies—an idea that originates in the theory of relational databases. In Section 2, we describe the key features of Haskell type classes that will be needed to understand the contributions of this paper. In Section 3, we use the design of a simple library of collection types to illustrate the problems that can occur with multiple parameter classes, and to motivate the introduction of functional dependencies. Further examples are provided in Section 4. Basic elements of the theory of functional dependencies are presented in Section 5, and are used to explain their role during type inference in Section 6. In Section 7, we describe some further opportunities for using dependency information, and then we conclude with some pointers to future work in Section 8.

2 Preliminaries: Type Classes in Haskell

This section describes the *class declarations* that are used to introduce new (single parameter) type classes in Haskell, and the *instance declarations* that are used to populate them. Readers who are already familiar with these aspects of Haskell should probably skip ahead to the next section. Those requiring more than the brief overview given here should refer to the Haskell report [11] or to the various tutorials and references listed on the Haskell website at `http://haskell.org`.

Class Declarations: A class declaration specifies the name for a class and lists the member functions that each type in the class is expected to support. The actual types in each class—which are normally referred to as the *instances* of the class—are described using separate declarations, as will be described below. For example, an *Eq* class, representing the set of equality types, might be introduced by the following declaration:

class $Eq\ a$ **where**
  $(==) :: a \to a \to Bool$

The type variable a that appears in both lines here represents an arbitrary instance of the class. The intended reading of the declaration is that, if a is a particular instance of Eq, then we can use the $(==)$ operator at type $a \to a \to Bool$ to compare values of type a.

Qualified Types: As we have already indicated, the restriction on the use of the equality operator is reflected in the type that is assigned to it:

$(==) :: Eq\ a \Rightarrow a \to a \to Bool$

Types that are restricted by a predicate like this are referred to as *qualified types* [4]. Such types will be assigned to any function that makes either direct

or indirect use of the member functions of a class at some unspecified type. For example, the functions:

$$member\ x\ xs = any\ (x ==)\ xs$$
$$subset\ xs\ ys\ = all\ (\backslash x \to member\ x\ ys)\ xs$$

will be assigned types:

$$member :: Eq\ a \Rightarrow a \to [a] \to Bool$$
$$subset\ \ :: Eq\ a \Rightarrow [a] \to [a] \to Bool.$$

Superclasses: Classes may be arranged in a hierarchy, and may have multiple member functions. The following example illustrates both with a declaration of the *Ord* class, which contains the types whose elements can be ordered using strict (<) and non-strict (<=) comparison operators:

class $Eq\ a \Rightarrow Ord\ a$ **where**
  $(<), (<=) :: a \to a \to Bool$

In this particular context, the ⇒ symbol should not be read as implication; in fact reverse implication would be a more accurate reading, the intention being that every instance of *Ord* is also an instance of *Eq*. Thus *Eq* plays the role of a *superclass* of *Ord*. This mechanism allows the programmer to specify an expected relationship between classes: it is the compiler's responsibility to ensure that this property is satisfied, or to produce an error diagnostic if it is not.

Instance Declarations: The instances of any given class are described by a collection of instance declarations. For example, the following declarations show how one might define equality for booleans, and for pairs:

instance *Eq Bool* **where**
  $x == y =$ **if** x **then** y **else** $not\ y$

instance $(Eq\ a,\ Eq\ b) \Rightarrow Eq\ (a,\ b)$ **where**
  $(x,\ y) == (u,\ v) = (x == u\ \texttt{\&\&}\ y == v)$

The first line of the second instance declaration tells us that an equality on values of types a and b is needed to provide an equality on pairs of type $(a,\ b)$. No such preconditions are need for the definition of equality on booleans. Even with just these two declarations, we have already specified an equality operation on the infinite family of types that can be constructed from *Bool* by repeated uses of pairing. Additional declarations, which may be distributed over many modules, can be used to extend the class to include other datatypes.

3 Example: Building a Library of Collection Types

One of the most commonly suggested applications for multiple parameter type classes is to provide uniform interfaces to a wide range of collection types [10].

Such types might be expected to offer ways to construct empty collections, to insert values, to test for membership, and so on. The following declaration, greatly simplified for the purposes of presentation, introduces a two parameter class *Collects* that could be used as the starting point for such a project:

```
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 collection 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 e 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 [2, 4]. For this reason, a Haskell system will reject any attempt to define or use such terms.

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 coll = insert x (insert y coll)
g coll     = f True 'a' coll
```

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 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, *coll*. If we hope 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. Thus the error in this code will not be detected at the point of definition, but only at the point of use, which might not even be in the same module. Obviously, we would prefer to avoid these problems, eliminating ambiguities, inferring more accurate types, and providing earlier detection of type errors.

3.1 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 $:: e \to c\ e \to c\ e$
  member $:: e \to c\ e \to \mathit{Bool}$

In fact this is precisely the approach taken by Okasaki [9], and by Peyton Jones [10], in more realistic attempts to build this kind of library. 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. Thus *Collects* is an example of a *constructor class* [6] in which the second parameter is a unary type *constructor*, replacing the nullary type parameter ce that was used in the original definition. This change avoids the immediate problems that we mentioned above:

- The *empty* operator has type $\mathit{Collects}\ e\ c \Rightarrow c\ e$, which is not ambiguous because both e and c appear on the right of the $\Rightarrow$ symbol.
- The function f is assigned a more accurate type:

$$f :: (\mathit{Collects}\ e\ c) \Rightarrow e \to e \to c\ e \to c\ e.$$

- The function g is now rejected, as required, with a type error 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. The reason that it works, at least intuitively, is that its two parameters are essentially independent of one another and so there is a good fit with the interpretation of *Collects* as a relatively unconstrained relation between types e and type constructors c.

Unfortunately, this version of the *Collects* class is not as general as the original class seemed to be. Only one of the four instances listed in Section 3 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. Some of the remaining instances can be

reworked to fit the constructor class framework by introducing dummy type and value constructors, as in the following example:

newtype *CharFun* e = *MkCharFun* $(e \to Bool)$
instance *Eq* $e \Rightarrow$ *Collects* e *CharFun* **where** ...

This approach, however, is not particularly attractive. It clutters up programs with the artificial type constructor *CharFun*, and with uses of the value constructor *MkCharFun* to convert between the two distinct but equivalent representations of characteristic functions. The workaround is also limited, and cannot, in general, deal with cases like the *BitSet* example, where the element type is fixed and not a variable e that we can abstract over.

3.2 Using Parametric Type Classes

Another alternative is to use *parametric type classes* [3] (PTC), with predicates of the form $ce \in Collects\ e$, meaning that ce is a member of the class *Collects* e. Intuitively, there is one type class *Collects* e for each choice of the e parameter. The definition of a parametric *Collects* class looks much like the original:

class $ce \in Collects\ e$ **where**
  empty :: ce
  insert :: $e \to ce \to ce$
  member :: $e \to ce \to Bool$

All of the instances declarations that we gave for the original *Collects* class in Section 3 can be adapted to the syntax of PTC, without introducing artificial type constructors. What makes it different from the two parameter class in Section 3 is the implied assumption that the element type e is uniquely determined by the collection type ce. A compiler that supports PTC must ensure that the declared instances of *Collects* do not violate this property. In return, it can use this information to avoid ambiguity and to infer more accurate types. For example, the type of *empty* is now $\forall e, ce.(ce \in Collects\ e) \Rightarrow ce$, and we do not need to treat this as being ambiguous because the unknown element type e is uniquely determined by ce.

Thus, PTC provides exactly the tools that we need to define and work with a library of collection classes. In our opinion, the original work on PTC has not received the attention that it deserves. In part, this may be because it was seen, incorrectly, as an alternative to constructor classes and not, more accurately, as an orthogonal extension. In addition, there has never been even a prototype implementation for potential users to experiment with.

3.3 Using Functional Dependencies

In this paper, we describe a generalization of parametric type classes that allows programmers to declare explicit *functional dependencies* between the parameters of a predicate. For example, we can achieve the same effects as PTC, with no

further changes in notation, by annotating the original class definition with a dependency $ce \rightsquigarrow e$, to be read as "ce uniquely determines e."

class *Collects* e $ce \mid ce \rightsquigarrow e$ **where**
  empty :: ce
  insert :: $e \to ce \to ce$
  member :: $e \to ce \to Bool$

More generally, we allow class declarations to be annotated with (zero or more) dependencies of the form $(x_1, \ldots, x_n) \rightsquigarrow (y_1, \ldots, y_m)$, where $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$ are type variables and $m, n > 0$[1]. Such a dependency is interpreted as an assertion that the y parameters are uniquely determined by the x parameters. Dependencies appear only in class declarations, and not in any other part of the language: the syntax for instance declarations, class constraints, and types is completely unchanged. For convenience, we allow the parentheses around a list of type variables in a dependency to be omitted if only a single variable is used.

This approach is strictly more general than PTC because it allows us to express a larger class of dependencies, including mutual dependencies such as $\{a \rightsquigarrow b, b \rightsquigarrow a\}$. It is also easier to integrate with the existing syntax of Haskell because it does not require any changes to the syntax of predicates.

By including dependency information, programmers can specify multiple parameter classes more precisely. To illustrate this, consider the following examples:

class C a b **where** ...
class D a $b \mid a \rightsquigarrow b$ **where** ...
class E a $b \mid a \rightsquigarrow b, b \rightsquigarrow a$ **where** ...

From the first declaration, we can tell only that C is a binary relation. The dependency $a \rightsquigarrow b$ in the second declaration tells us that D is not just a relation, but actually a (partial) function. From the two dependencies in the last declaration, we can see that E represents a (partial) one-one mapping.

The compiler is responsible for ensuring that the instances in scope at any given point are consistent with any declared dependencies[2]. For example, the following declarations cannot appear together 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

[1] For practical reasons, a slightly different syntax is used for dependencies in the current prototype implementation, details of which are included in the distribution.

[2] Superclass declarations are handled in a similar way, leaving the compiler to ensure that every instance of a given class is also an instance of any superclasses.

declaration of the form **instance** $\ldots \Rightarrow D\ t\ s$ **where** $\ldots$, 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.

4 Further Examples

This section presents two additional examples to show how the use of functional dependencies can allow us to give more accurate specifications and to make more practical use of multiple parameter type classes.

Arithmetic Operations The Haskell prelude treats arithmetic functions like addition $(+)$ and multiplication $(*)$ as functions of type $Num\ a \Rightarrow a \to a \to a$, which means that the result will always be of the same type as the arguments. A more flexible approach would allow different argument types so that we could add two *Int* values to get an *Int* result, or add an *Int* to a *Float* to get a *Float* result. This more flexible approach can be coded as follows:

class $Add\ a\ b\ c \mid (a, b) \rightsquigarrow c$ **where** $(+) :: a \to b \to c$
class $Mul\ a\ b\ c \mid (a, b) \rightsquigarrow c$ **where** $(*) :: a \to b \to c$

instance $Mul\ Int\ Int\ Int$ **where** $\ldots$
instance $Mul\ Int\ Float\ Float$ **where** $\ldots$
instance $Mul\ Float\ Int\ Float$ **where** $\ldots$
instance $Mul\ Float\ Float\ Float$ **where** $\ldots$

In a separate linear algebra package, we might further extend our classes with arithmetic operations on vectors and matrices:

instance $Mul\ a\ b\ c \Rightarrow Mul\ a\ (Vec\ b)\ (Vec\ c)$ **where** $\ldots$
instance $Mul\ a\ b\ c \Rightarrow Mul\ a\ (Mat\ b)\ (Mat\ c)$ **where** $\ldots$
instance $(Mul\ a\ b\ c, Add\ c\ c\ d)$
$\Rightarrow Mul\ (Mat\ a)\ (Mat\ b)\ (Mat\ d)$ **where** $\ldots$

Without dependency information, we quickly run into problems with ambiguity. For example, even simple expressions like $(1 * 2) * 3$ have ambiguous types:

$(1 * 2) * 3 :: (Mul\ Int\ Int\ a, Mul\ a\ Int\ b) \Rightarrow b.$

Using the dependencies, however, we can determine that $a = Int$, and then that $b = Int$, and so deduce that the expression has type *Int*. This example shows that it can be useful to allow multiple types on the left hand side of a dependency.

Finite Maps A *finite map* is an indexed collection of elements that provides operations to lookup the value associated with a particular index, or to add a new binding. This can be described by a class:

class $FiniteMap\ i\ e\ fm \mid fm \rightsquigarrow (i, e)$ **where**
$emptyFM :: fm$
$lookup \quad :: i \to fm \to Maybe\ e$
$extend \quad :: i \to e \to fm \to fm$

Here, *fm* is the finite map type, which uniquely determines both the index type i and the element type e. Association lists, functions, and arrays all fit naturally into this framework. We can also use a bit set as an indexed collection of booleans:

instance $(Eq\ i) \Rightarrow FiniteMap\ i\ e\ [(i,\ e)]$ **where** ...
instance $(Eq\ i) \Rightarrow FiniteMap\ i\ e\ (i \rightarrow e)$ **where** ...
instance $(Ix\ i) \Rightarrow FiniteMap\ i\ e\ (Array\ i\ e)$ **where** ...
instance *FiniteMap Int Bool BitSet* **where** ...

This is a variation on the treatment of collection types in Section 3, and, if the dependency is omitted, then we quickly run into very similar kinds of problem. We have included this example here to show that it can be useful to allow multiple types on the right hand side of a dependency.

5 Relations and Functional Dependencies

In this section, we provide a brief primer on the theory of relations and functional dependencies, as well as a summary of our notation. These ideas were originally developed as a foundation for relational database design [1]. They are well-established, and more detailed presentations of the theory, and of useful algorithms for working with them in practical settings, can be found in standard textbooks on the theory of databases [8]. A novelty of the current paper is in applying them to the design of a type system.

5.1 Relations

Following standard terminology, a *relation* R over an indexed family of sets $\{D_i\}_{i \in I}$ is just a set of *tuples*, each of which is an indexed family of values $\{t_i\}_{i \in I}$ such that $t_i \in D_i$ for each $i \in I$. More formally, R is just a subset of $\Pi i \in I.D_i$, where a tuple $t \in (\Pi i \in I.D_i)$ is a function that maps each index value $i \in I$ to a value $t_i \in D_i$ called the ith component of t. In the special case where $I = \{1, \ldots, n\}$, this reduces to the familiar special case where tuples are values $(t_1, \ldots, t_n) \in D_1 \times \ldots \times D_n$. If $X \subseteq I$, then we write t_X, pronounced "t at X", for the restriction of a tuple t to X. Intuitively, t_X just picks out the values of t for the indices appearing in X, and discards any remaining components.

5.2 Functional Dependencies

In the context of an index set I, a *functional dependency* is a term of the form $X \rightsquigarrow Y$, read as "X determines Y," where X and Y are both subsets of I. If a relation satisfies a functional dependency $X \rightsquigarrow Y$, then the values of any tuple at Y are uniquely determined by the values of that tuple at X. For example, taking $I = \{1, 2\}$, relations satisfying $\{\{1\} \rightsquigarrow \{2\}\}$ are just partial functions from D_1 to D_2, while relations satisfying $\{\{1\} \rightsquigarrow \{2\}, \{2\} \rightsquigarrow \{1\}\}$ are partial, injective functions from D_1 to D_2.

If F is a set of functional dependencies, and $J \subseteq I$ is a set of indices, then the *closure* of J with respect to F, written J_F^+ is the smallest set such that $J \subseteq J_F^+$, and that, if $(X \rightsquigarrow Y) \in F$, and $X \subseteq J_F^+$, then $Y \subseteq J_F^+$. For example, if $I = \{1, 2\}$, and $F = \{\{1\} \rightsquigarrow \{2\}\}$, then $\{1\}_F^+ = I$, and $\{2\}_F^+ = \{2\}$. Intuitively, the closure J_F^+ is the set of indices that are uniquely determined, either directly or indirectly, by the indices in J and the dependencies in F. Closures like this are easy to compute using a simple fixed point iteration.

6 Typing with Functional Dependencies

This section explains how to extend an implementation of Haskell to deal with functional dependencies. In fact the tools that we need are obtained as a special case of improvement for qualified types [5]. We will describe this briefly here; space restrictions prevent a more detailed overview. To simplify the presentation, we will assume that there is a set of indices (i.e., parameter names), written I_C, and a corresponding set of functional dependencies, written F_C, for each class name C. We will also assume that all predicates are written in the form $C\ t$, where t is a tuple of types indexed by I_C. This allows us to abstract away from the order in which the components are written in a particular implementation.

The type system of Haskell can be described using judgements of the form $P \mid A \vdash E : \tau$. Each such judgement represents an assertion that an expression E can be assigned a type τ, using the assumptions in A to type any free variables, and providing that the predicates in P are satisfied. When we say that a set of predicates is satisfied, we mean that they are all implied by the class and instance declarations that are in scope at the corresponding point in the program. For a given A and E, the goal of type inference is to find the most general choices for P and τ such that $P \mid A \vdash E : \tau$. If successful, we can infer a principal type for E by forming the qualified type $P \Rightarrow \tau$—without looking at the predicates in P—and then quantifying over all variables that appear in $P \Rightarrow \tau$ but not in A.

One of the main results of the theory of improvement is that we can apply *improving substitutions* to the predicate set P at any point during type inference (and as often as we like), without compromising on a useful notion of principal types. Intuitively, an improving substitution is just a substitution that can be applied to a particular set of predicates without changing its satisfiability properties. To make this more precise, we will write $\lfloor P \rfloor$ for the set of satisfiable instances of P, which is defined by:

$$\lfloor P \rfloor = \{SP \mid S \text{ is a substitution and the predicates in } SP \text{ are satisfied}\}.$$

In this setting, we say that S is an improving substition for P if $\lfloor P \rfloor = \lfloor SP \rfloor$, and if the only variables involved in S that do not also appear in P are 'new' or 'fresh' type variables. From a practical perspective, this simply means that the subsitution will not change the set of environments or the set of types at which a given value can be used. The restriction to new variables is necessary to avoid conflicts with other type variables that might already be in use.

Improvement cannot play a useful role in a standard Haskell type system: The language does not restrict the choice of instances for any given type class, and hence the only improving substitions that we can obtain are equivalent to an identity substitution. With the introduction of functional dependencies, however, we do restrict the set of instances that can be defined, and this leads to opportunities for improvment. For example, by prohibiting the definition of instances of the form *Collects* a $[b]$ where $a \neq b$, we know that we can use an improving substitution $[a/b]$ and map any such predicate into the form *Collects* a $[a]$.

6.1 Ensuring that Dependencies are Valid

Our first task is to ensure that all declared instances for a class C are consistent with the functional dependencies in F_C. For example, suppose that we have an instance declaration for C of the form:

instance $\ldots \Rightarrow C\ t$ **where** $\ldots$

Now, for each $(X \rightsquigarrow Y) \in F_C$, we must ensure that $TV(t_Y) \subseteq TV(t_X)$ or otherwise the elements of t_Y might not be uniquely determined by the elements of t_X. (The notation $TV(X)$ refers to the set of type variables appearing free in the object X.) A further restriction is needed to ensure pairwise compatibility between instance declarations for C. For example, if we have a second instance:

instance $\ldots \Rightarrow C\ s$ **where** $\ldots$,

and a dependency $(X \rightsquigarrow Y) \in F_C$, then we must ensure that $t_Y = s_Y$ whenever $t_X = s_X$. In fact, on the assumption that the two instances will normally contain type variables—which could later be instantiated to more specific types—we will actually need to check that: for all (kind-preserving) substitutions S, if $St_X = Ss_X$, then $St_Y = Ss_Y$. It is easy to see that this test can be reduced to checking that, if t_X and s_X have a most general unifier U, then $Ut_Y = Us_Y$. This is enough to guarantee that the declared dependencies are satisfied. For example, the instance declarations in Section 3 are consistent with the dependency $ce \rightsquigarrow e$.

6.2 Improving Inferred Types

There are two ways that a dependency $(X \rightsquigarrow Y) \in F_C$ for a class C can be used to help infer more accurate types:

- If we have predicates $(C\ t)$ and $(C\ s)$ with $t_X = s_X$, then t_Y and s_Y must be equal.
- Suppose that we have an inferred predicate $C\ t$, and an instance:

  instance $\ldots \Rightarrow C\ t'$ **where** $\ldots$

  If $t_X = St'_X$, for some substitution S (which could be calculated by one-way matching), then t_Y and St'_Y must be equal.

In both cases, we can use unification to ensure that the equalities are satisfied, and to calculate a suitable improving substitution [5]. If unification fails, then we have detected a type error. Note that we will, in general, need to iterate this process until no further opportunities for improvement can be found.

6.3 Detecting Ambiguity

As mentioned in Section 3, we cannot guarantee a well-defined semantics for any function that has an ambiguous type. With the standard definition, a type of the form $(\forall a_1 \ldots \forall a_n.P \Rightarrow \tau)$ is ambiguous if $(\{a_1, \ldots, a_n\} \cap TV(P)) \not\subseteq TV(\tau)$, indicating that one of the quantified variables a_i appears in $TV(P)$ but not in $TV(\tau)$. Our intuition is that, if there is no reference to a_i in the body of the type, then there will be no way to determine how it should be bound when the type is instantiated. However, in the presence of functional dependencies, there might be another way to find the required instantiation of a_i. We need not insist that every $a \in TV(P)$ is mentioned explicitly in τ, so long as they are all uniquely determined by the variables in $TV(\tau)$.

The first step to formalizing this idea is to note that every set of predicates P induces a set of functional dependencies F_P on the type variables in $TV(P)$:

$$F_P = \{\, TV(t_X) \rightsquigarrow TV(t_Y) \,|\, (C\ t) \in P,\ (X \rightsquigarrow Y) \in F_C \,\}.$$

This has a fairly straightforward reading: if all of the variables in t_X are known, and if $X \rightsquigarrow Y$, then the components of t at X are also known, and hence so are the components, and thus the type variables, in t at Y.

To determine if a type $(\forall a_1 \ldots \forall a_n.P \Rightarrow \tau)$ is ambiguous, we calculate the set of dependencies F_P, and then take the closure of $TV(\tau)$ with respect to F_P to obtain the set of variables that are determined by τ. The type is ambiguous only if there are variables a_i in P that are not included in this closure. More concisely, the type is *ambiguous* if, and only if $(\{a_1, \ldots, a_n\} \cap TV(P)) \not\subseteq (TV(\tau))^+_{F_P}$.

On a related point, we note that current implementations of Haskell are required to check that, in any declaration of the form **instance** $P \Rightarrow C\ t$ **where** ..., only the variables appearing in t can be used in P (i.e., we must ensure that $TV(P) \subseteq TV(t)$). In light of the observations that have been made in this section, we can relax this to require only that $TV(P) \subseteq (TV(t))^+_{F_P}$. Thus P may contain variables that are not explicitly mentioned in t, provided that they are still determined by the variables in t.

6.4 Generalizing Inferred Types

In a standard Hindley-Milner type system, principal types are computed using a process of generalization. Given an inferred but unquantified type $P \Rightarrow \tau$, we would normally just calculate the set of type variables $T = TV(P \Rightarrow \tau)$, over which we might want to quantify, and the set of variables $V = TV(A)$ that are fixed in the current assumptions A, and then quantify over any variables in the difference, $T \setminus V$. In the presence of functional dependencies, however, we must be a little more careful: a variable a that appears in T but not in V may still need to be treated as a fixed variable if it is determined by V. To account for this, we should only quantify over the variables in $T \setminus V^+_{F_P}$.

7 Putting a Name to Functional Dependencies

The approach described in this paper provides a way for programmers to indicate that there are dependencies between the parameters of a type class, but stops short of giving those dependencies a name. To illustrate this point, consider the following pair of class declarations:

class $U\ a\ b \mid a \rightsquigarrow b$ **where** ...
class $U\ a\ b \Rightarrow V\ a\ b$ **where** ...

From the first declaration, we know that there is a dependency between the parameters of U; should there not also be a dependency between the parameters of V, inherited from its superclass U? Such a dependency could be added by changing the second declaration to:

class $U\ a\ b \Rightarrow V\ a\ b \mid a \rightsquigarrow b$ **where** ...

but this tells only part of the story. For example, given two predicates $U\ a\ b$ and $V\ a\ c$, nothing in the rules from Section 6 will allow us to infer that $b = c$. Let us return to the dependency on U and give a name to it by writing u for the function that maps each a to the b that it determines. This might even be made explicit in the syntax of the language by changing the declaration to read:

class $U\ a\ b \mid u :: a \rightsquigarrow b$ **where** ...

Now we can change the declaration of V again to indicate that it inherits the same dependency u:

class $U\ a\ b \Rightarrow V\ a\ b \mid u :: a \rightsquigarrow b$ **where** ...

Now, given the predicates $U\ a\ b$ and $V\ a\ c$, we can infer that $b = u\ a = c$, as expected. It is not yet clear how useful this particular feature might be, or whether it might be better to leave the type checker to infer inherited dependencies automatically, without requiring the programmer to provide names for them. The current prototype includes an experimental implementation of this idea (without making dependency names explicit), but the interactions with other language features, particularly overlapping instances, are not yet fully understood. Careful exploration of these issues is therefore a topic for future work. However, the example does show that there are further opportunities to exploit dependency information that go beyond the ideas described in Section 6.

8 Conclusions and Future Work

The ideas described in this paper have been implemented in the latest version of the Hugs interpreter [7], and seem to work well in practice. Pleasingly, some early users have already found new applications for this extension in their own work, allowing them to overcome problems that they had previously been unable to fix. Others have provided feedback that enabled us to discover places where further use of dependency information might be used, as described in Section 7.

In constructing this system, we have used ideas from the theory of relational databases. One further interesting area for future work would be to see if other ideas developed there could also be exploited in the design of programming language type systems. Users of functional languages are, of course, accustomed to working with parameterized datatypes. Functional dependencies provide a way to express similar relationships between types, without being quite so specific. For example, perhaps similar ideas could be used in conjunction with existential types to capture dependencies between types whose identities have been hidden?

Acknowledgments

I would like to thank my colleagues at OGI for their interest in this work. Particular thanks go to Jeff Lewis for both insight and patches, and to Lois Delcambre for explaining the role that functional dependencies play in database theory.

References

[1] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Cong.*, Geneva, Switzerland, 1974.
[2] S. M. Blott. *An approach to overloading with polymorphism.* PhD thesis, Department of Computing Science, University of Glasgow, September 1991.
[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (extended abstract). In *ACM conference on LISP and Functional Programming*, San Francisco, CA, June 1992.
[4] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.
[5] M. P. Jones. Simplifying and improving qualified types. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995.
[6] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.
[7] M. P. Jones and J. C. Peterson. *Hugs 98 User Manual*, September 1999.
[8] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.
[9] C. Okasaki. *Edison User's Guide*, May 1999.
[10] S. Peyton Jones. Bulk types with class. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.
[11] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, February 1999.
[12] S. Peyton Jones and J. Hughes (editors). *Standard libraries for the Haskell 98 programming language*, February 1999.
[13] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.
[14] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.