Language and Program Design for Functional Dependencies

Mark P. Jones
Portland State University
mpj@cs.pdx.edu

Iavor Diatchki
Galois, Inc.
iavor.diatchki@gmail.com

Abstract

Eight years ago, functional dependencies, a concept from the theory of relational databases, were proposed as a mechanism for avoiding common problems with multiple parameter type classes in Haskell. In this context, functional dependencies give programmers a means to specify the semantics of a type class more precisely, and to obtain more accurate inferred types as a result. As time passed, however, several issues were uncovered—both in the design of a language to support functional dependencies, and in the ways that programmers use them—that led some to search for new, better alternatives.

This paper focusses on two related aspects of design for functional dependencies: (i) the design of language/type system extensions that implement them; and (ii) the design of programs that use them. Our goal is to clarify the issues of what functional dependencies are, how they should be used, and how the problems encountered with initial proposals and implementations can be addressed.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Design, Languages

Keywords Functional dependencies, Type functions, Type inference, Relational databases, Qualified types, Haskell

1. Introduction

Since its origins nearly four decades ago, relational database theory has relied on the use of functional dependencies to document and characterize the semantic structure of database tables and to formalize concepts such as database normalization [1, 6, 7, 8]. More recently, around eight years ago, the Hugs and GHC implementations of Haskell introduced experimental support for annotating type classes with functional dependencies [18], and for using the information that they provide to "improve" the types obtained by type inference [16]. This facility allowed programmers to avoid many of the ambiguities and inaccurate typing problems that plagued early implementations of multiple parameter type classes and quickly became one of the more widely used extensions of Haskell 98.

Early users reported "Fun with Functional Dependencies" [13], but there have also been some problems. Confounded by misunderstandings, awkward notation, and buggy implementations—some resulting from unexplored interactions with other experimental features—the Haskell community began to explore alternatives to functional dependencies. Examples include proposals for a more 'functional notation' [25] and, most recently, for associated types [4] and type families [26]. If it is any measure of frustration, there has even been a (light-hearted) call for "Death to functional dependencies" [21]! As the community looks to the finalization of a new language standard, currently dubbed Haskell', there is still uncertainty in deciding which of the main contenders—functional dependencies or type families—should be adopted.

It is a mistake, however, to judge functional dependencies by the original paper [18], or by the mostly widely used implementations, because these have not been updated to reflect all of the lessons that have been learned. The goal of this paper is to provide a more current perspective, with a focus on two aspects of design: (i) the design of language extensions to support functional dependencies; and (ii) the design of programs that use functional dependencies. As part of this, we describe problems with the original proposal and implementations, and we suggest ways in which they can be addressed. We hope that this will help to fill some gaps and to highlight less well-known details so that the strengths and weaknesses of different proposals can be better understood. It is not our intention to advocate for (or against) adoption or use of functional dependencies in Haskell', but rather to clarify key issues that arise in this part of the design space for Haskell-like type systems.

An unusual feature of this paper is the inclusion of two fairly substantial appendices that provide technical summaries of "Functional Dependencies in Database Theory" (Appendix A) and "Improving Qualified Types" (Appendix B). We include this material because our experience suggests that users of functional dependencies in Haskell are not always familiar with the technical foundations on which they are based. For example, the authors of one published paper [27] asserted that "functional dependencies have never been formalized" and that relational database theory "has almost nothing to do with" functional dependencies for Haskell-style type classes. In fairness, some parts of their paper discuss new ideas that go beyond the formalization that is provided by the functional dependencies of database theory or the theory of improvement for qualified types. Taken at face value, however, their comments are too strong and potentially confusing to those who are trying to understand how different proposals relate to one another. We hope that the appendices, written specifically for this paper, will provide readers with a review/reference for these foundations without overly distracting from our main topic.

The remaining sections of this paper are as follows. In Section 2, we summarize background material including the development of type classes, the problems that led to the introduction of functional dependencies, and the formalization of a type system that supports functional dependencies. Some of the biggest problems that Haskell programmers encounter with functional dependencies have to do with the reliance on a relational notation of constraints. In Section 3, we explain a simple, syntactic technique that

Haskell'08, September 25, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-064-7/08/09... $5.00

can be used to address these problems by providing a lightweight, functional notation. Other issues that have caused problems with functional dependencies are summarized in Section 4. In Section 5, we turn our attention to the construction of programs that use functional dependencies, drawing inspiration from the theory of relational databases to suggest some guidelines for good design. Finally, we conclude with some brief comments in Section 6.

2. Background

Two of the original design goals for Haskell were "to reduce unnecessary diversity in functional programming languages" and to produce a design "based on ideas that enjoy a wide consensus" [14, 22]. The Haskell committee, however, quickly realized that there was no consensus solution for dealing with the so-called *ad-hoc polymorphism* that was needed to deal with key features of the language such as basic arithmetic and comparison operators, as well as the `show` function that converts values into printable strings. The challenge in each of these cases is to find a mechanism that allows the operators to be used with many different (but not necessarily all) types *without* requiring the use of a different symbol in each case; in short, what was needed was a principled and systematic way to support *overloading*.

The solution came in the form of *type classes*, following an original suggestion by Philip Wadler and then a more formal development by Wadler and Blott [28] and Blott [3]. A type class pairs a set of types, called the *instances* of the class, with an associated family of *member* operations that are defined for each instance. Using Haskell syntax, each type class is introduced by a declaration like the following standard example for the set of equality types:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y       = not (x==y) -- default implementation
```

This declaration provides a name, `Eq`, for the class. The parameter, `a`, represents an arbitrary instance that is used in the types of the `(==)` and `(/=)` operators to specify the types of the operators that should be provided for each instance. These instances are specified separately via a collection of instance declarations that may be distributed across the source code of a program or library. For example, the definition of equality on a primitive type like `Int` might be implemented by a primitive function called `primEqInt`, and wired into the `Eq` class using a definition like the following:

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

For parameterized types, like tuples for example, equality is often defined in terms of the equality operations on the parameter types:

```
class (Eq a, Eq b) => Eq (a, b) where
  (p,q) == (x,y)  =  (p==x) && (q==y)
```

The clause `(Eq a, Eq b) => Eq (a, b)` in the first line specifies that, if `a` is an equality type and `b` is an equality type, then the tuple type `(a,b)` will also be an equality type. The second line provides evidence for this by showing how the equality operation on pairs (the occurrence of `==` on the left hand side of the definition) can be obtained from the equality operations on `a` and `b` (the two occurrences of `==` on the right hand side, respectively).

2.1 A Semantics for Class and Instance Declarations

A compiler/type checker for Haskell uses declarations like these to answer (at least) two different kinds of question about a class:

1. Is a given type `t` an instance of the class?
2. If `t` is an instance of the class, then what are the corresponding implementations of the members?

We will identify the first of these questions with the task of providing a semantics for the name `Eq` as a set of types, and the second with the task of providing a semantics for the name `(==)` as a mapping from instances to implementations. Both are important but, in this paper, we focus on the first activity. For example, a natural way to give a meaning to `Eq` given only the declarations shown above is as the smallest set of types that satisfies the following equation:

$$\texttt{Eq} = \{\texttt{Int}\} \cup \{\, \texttt{(a,b)} \mid \texttt{a} \in \texttt{Eq},\, \texttt{b} \in \texttt{Eq}\,\}.$$

Informally, this gives us the following interpretation for `Eq`:

$$\begin{aligned}\texttt{Eq} = \{\, &\texttt{Int},\ \texttt{(Int,Int)},\ \texttt{(Int,(Int,Int))},\\ &\texttt{((Int,Int),Int)},\ \texttt{((Int,Int),(Int,Int))},\ \ldots\}.\end{aligned}$$

Alternatively, we can interpret each instance declaration as contributing a rule to a proof system for deriving statements of the form `Eq t` to indicate that `t` is an instance of `Eq`. For the instance declarations above, we have two proof rules: $\Vdash \texttt{Eq Int}$ and $\texttt{Eq a}, \texttt{Eq b} \Vdash \texttt{Eq (a,b)}$ These two approaches provide what one might consider as semantic and syntactic methods, respectively, for defining the meaning of a type class. Of course, they are closely related because any demonstration that a type `t` is an instance of the set represented by `Eq` corresponds directly to a proof that $\Vdash \texttt{Eq t}$.

2.2 Multiple Parameter Classes

Right from the beginning, there was considerable interest in extending the type class mechanisms of Haskell to support multiple parameters. Wadler and Blott [28], for example, offered the following code fragment as a hint at how multiple parameters might be used, in this case, to define an overloaded coercion operator:

```
class Coerce a b where
  coerce :: a -> b
instance Coerce Int Float where
  coerce = convertIntToFloat
```

Many examples like this were posted in the early days of the Haskell mailing list. There was excitement about how they might be used in practical Haskell programs, and considerable interest also in modifying one of the nascent Haskell implementations to support them. Unfortunately, the arrival of the first such implementation [19] brought with it the disappointing realization that many of these examples did not work as hoped! For example, the simple expression `coerce 0`, which might have been expected to produce a `Float` result of `0.0` from its argument `0` of type `Int`, instead produced a term of type `(Coerce a b, Num a) => b`. This type is verbose and, more seriously, *ambiguous* because the type variable `a` that is mentioned on the left of the `=>` symbol does not appear on the right-hand side; standard results guaranteeing *coherence* (i.e., a well-defined semantics for overloaded functions) do not hold for functions with ambiguous types [3, 15]. This problem can be traced to the way that Haskell overloads numeric literals, allowing them to be treated as having any numeric type, and it can be fixed by adding a type annotation to the original expression to obtain `coerce (0::Int)`. Even with this modification, however, the most general type of the expression is `(Coerce Int b) => b`, which is again more complicated than we had intended. A further type annotation can fix this: the expression `coerce (0::Int) :: Float` will type check with type `Float`, but at this point, with all the added type annotations, it would have been easier simply to have typed `convertIntToFloat 0` (or just `(0::Float)`, of course). In short, based on examples like this, it seemed that there would be little practical benefit to multiple parameter type classes because there was too little coupling between the parameters to obtain useful types without resorting to fairly heavy use of type annotations.

Experiences like these discouraged wide use of multiple parameter type classes, but there were some notable successes. Work on modular interpreters, for example, used multiple parameter classes, without problems, to implement a form of subtyping, and to categorize monads according to the computational features that they support [20, 23]. The following fragment illustrates this: `MonadState m s` indicates that monad `m` has a state component of type `s` that can be modified using the `update` function:

```
class Monad m => MonadState m s where
  update :: (s -> s) -> m s
```

In this case, the loose coupling between the two parameters is entirely appropriate: it is possible that a single monad `m` may have multiple state components `s`, and it is also possible that a single state component `s` may be used in multiple monads `m`[1].

2.3 A Semantics for Multiple Parameter Classes

The previous example hints that the `MonadState` class represents a relation between monads and state types. In fact, it is natural to view a general *n*-parameter type class as an *n*-place relation on types, that is, as a set of *n*-tuples on types. This even handles single parameter classes like `Eq` as a special case, identifying individual instance types with corresponding 1-tuples. For example, consider the following declarations that define a `plus` operator for adding two arguments, each of which may be either an `Int` or a `Float`:

```
class Plus a b c where
  plus :: a -> b -> c
instance Plus Int   Int   Int   where ...
instance Plus Int   Float Float where ...
instance Plus Float Int   Float where ...
instance Plus Float Float Float where ...
```

As a relation, the resulting `Plus` class can be viewed either as a set of 3-tuples, or, equivalently, pictured in a simple tabular form:

$\{$ (Int,Int,Int),
(Int,Float,Float),
(Float,Int,Float),
(Float,Float,Float) $\}$

a	b	c
Int	Int	Int
Int	Float	Float
Float	Int	Float
Float	Float	Float

As another example, consider the following (simplified) attempt to describe "collection types" as a two parameter class; `c` is the type of the collection and `e` is the type of the elements that it contains:

```
class Elem c e where
  insert :: e -> c -> c
  toList :: c -> [e]

instance Elem BitVector Char where ...
instance Eq e => Elem [e] e where ...
instance Ord e => Elem (BST e) e where ...
```

These declarations introduce the class `Elem` with instances for `BitVector` (as a collection of `Char` values) and for lists and binary search trees (the type `BST e`) of elements, provided that the element types have an associated equality or ordering, respectively. Once again, we can picture (a fragment of) the resulting relation as a table, this time with a column for each of the parameters `c` and `e`:

[1] Readers familiar with the monad transformer libraries in Haskell can construct a monad with two distinct state components using a type of the form `StateT s1 (StateT s2 m)`, or two different monads with a common state type using `StateT s m1` and `StateT s m2`.

c	e
BitVector	Char
[Int]	Int
[[Int]]	[Int]
⋮	⋮
BST Int	Int
BST [Int]	[Int]
⋮	⋮

We have drawn this table with two gaps. The first is a place holder for further `Elem` instances using lists, while the second is for instances using `BST` (and also leaves open the possibility of extending the program with further instance declarations).

A complete Haskell program includes many class and instance declarations, possibly with interesting recursion or dependencies between them. Together, these define a set of relations (one for each class) either as the solution of a set of (mutually recursive) equations or, equivalently, as a proof system generated from the instance declarations. A key insight here is that this set of relations *can be interpreted, viewed, and manipulated as a relational database*. In particular, this means that we can leverage interesting results and tools from database theory (such as those in Appendix A) to help us specify and use multiple parameter type classes more effectively.

We will begin to see the benefits of this approach shortly when we introduce and apply functional dependencies to `Plus` and `Elem`. We close this section, however, by observing that the tables we obtain as the semantics of type classes do differ in two respects from the tables of a conventional relational database—although neither is significant for the purposes of this paper. First, the values in each column are type expressions, so they carry some structure and are not just atomic values. Of course, we can choose to ignore such structure so that the tables can still be considered to be in first normal form. Second, even though each such table is generated from a finite set of instance declarations, the result may still include infinitely many rows. From that perspective, it would be even more accurate to characterize the semantics of type classes as a form of *deductive* rather than just a simpler, relational database.

2.4 Leveraging Functional Dependencies

The `Plus` and `Elem` examples described in the previous section suffer from the usual problems that we encounter with multiple parameter classes. For example, a seemingly simple function to add three numbers, `plus3 x y z = plus x (plus y z)`, receives a surprisingly complicated and, more seriously, ambiguous type (because the variable `d` mentioned on the left of the `=>` symbol does not appear on the right):

```
(Plus a d e, Plus b c d) => a -> b -> c -> e
```

The next definition illustrates difficulties with the `Elem` class:

```
insert2 x y c = insert x (insert y c)
```

This function attempts to insert two values, `x` and `y`, into a single collection, `c`, and the inferred type that we obtain is as follows:

```
insert2 :: (Elem c a, Elem c b) => a -> b -> c -> c
```

The key point is that this allows the two arguments to have different types. There is nothing fundamentally wrong with that, and it is certainly possible to define collections that contain more than one type of value. For example, a `[Either Bool Char]` list could be used to represent a collection of Boolean and character values mixed together, and we might expect to modify such a collection using a function `insert2 True 'a'`. On the other hand, it would also be reasonable to expect that any given collection contains only one type of value, in which case `insert2 True 'a'` should be treated as a type error. The programmer who created the `Elem` class may have had a particular interpretation in mind, but there is no way to express this in Haskell. Instead, the standard approach has been to adopt the former, more general interpretation in which multiple parameter classes can be instantiated to essentially arbitrary relations. What can be done if this was not the intended interpretation?

As it happens, the tables for `Plus` and `Elem` exhibit some interesting structure, and are not just arbitrary relations on types. For example, we can see that the result type, `c`, of a `plus` operation is uniquely determined by the types, `a` and `b`, of its arguments. In a similar way, the element type, `e` in each `Elem` instance is uniquely determined by the collection type, `c`. In database terminology, we might characterize these observations by means of functional dependencies (Section A.2): $\{\texttt{a,b}\} \leadsto \{\texttt{c}\}$ and $\{\texttt{c}\} \leadsto \{\texttt{e}\}$, respectively. The key idea of Jones [18] was to allow dependencies like these to be stated as annotations on class declarations as follows[2]:

```
class Plus a b c | a b -> c where
  plus :: a -> b -> c

class Elem c e | c -> e where
  insert :: e -> c -> c
  toList :: c -> [e]
```

By writing these annotations, the programmer documents the expected structure of each class more precisely, and, in so doing, restricts the ways in which the class may be populated. For example, with the functional dependency annotations in place, the compiler should now raise an error if we attempt to add the following instance declaration to the four described previously:

```
instance Plus Int Int Float where ...
```

The reason that we cannot allow this is because it conflicts with the previously declared `Plus Int Int Int` instance. That is, if we allowed a program to contain both instances, then the table for `Plus` would contain two rows with the same `a` and `b` entries but distinct values for `c`, violating the asserted functional dependency.

This shows that there is a price to pay for annotating a class with dependencies, both for programmers (because it limits the ways in which instances can be defined) and for implementations (because extra checks are required to ensure that dependencies are not violated). However, there is also a significant payback because information about dependencies can also be used to infer more accurate types. For example, given two predicates `Elem c a` and `Elem c b` with the same collection type, `c`, we can immediately infer from the functional dependency that `a`=`b`. This simple "improvement" (see Appendix B) allows us to infer a more specific type for `insert2`:

```
insert2 :: (Elem c e) => e -> e -> c -> c
```

An immediate consequence is that `insert2 True 'a'` will now trigger a type error. Also, while the `plus3` function still receives the same complicated type, it is no longer necessary to consider it ambiguous. Although `d` does not appear on the right hand side of the type, the `Plus b c d` constraint that appears in the type is enough to ensure that `d` will be uniquely determined by `b` and `c`, thanks to the dependency on `Plus`. It follows that there is no ambiguity because both of those variables do appear on the right of the `=>` symbol. For example, if we apply `plus3` to three arguments of type `Int`, then we can infer that `d`=`Int` (applying the dependency to the declared instance `Plus Int Int Int` and the inferred predicate `Plus Int Int d`), and hence, by a similar argument, that `e`=`Int`, without any need for further type annotations.

[2] There is no syntactic ambiguity in using the `->` symbol here to express dependencies. With hindsight, however, it might be better to use a different notation so that dependencies are not so easily confused with regular types.

Since their introduction, functional dependencies have been used in a wide range of examples like these to avoid problems with ambiguity and imprecise types in practical uses of multiple parameter classes. The main purpose of a dependency annotation is to let the designer of a class specify its semantics more precisely. Neither one of the two definitions that we have shown for `Elem`, for example, is intrinsically 'better' than the other. However, by including support for dependency annotations in the language, we are allowing designers to consider and document the choice between different alternatives explicitly as part of the source text.

2.5 Formalizing Functional Dependencies

This section summarizes the formal treatment of functional dependencies for Haskell and assumes familiarity with the notation and results of Appendices A and B. Readers who are less interested in theoretical aspects are encouraged to skip directly to Section 2.6.

We begin with some notation. We assume that there is a collection of class names C, each with an associated set of parameter names/indices, I_C, and an associated set of functional dependencies, F_C. We also assume that predicates are written in the form $C\ t$, where t is a tuple of types indexed by I_C. The notation F_C can be generalized to describe the set of *induced functional dependencies*, F_P, on the type variables, $TV(P)$, of a predicate set P:

$$\{\, TV(t_X) \leadsto TV(t_Y) \mid (C\ t) \in P,\ (X \leadsto Y) \in F_C \,\}.$$

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

2.5.1 Validating Instance Declarations

Our first task is to formalize conditions to ensure that the declared instances for each class C are compatible with the declared dependencies in F_C. This has two parts:

- *Covering*: To ensure that `instance` P `=>` $C\ t$ `where ...` is valid, we must check that $TV(t_Y) \subseteq (TV(t_X))^+_{F_P}$ for each $(X \leadsto Y) \in F_C$. Intuitively, this ensures that values for any variables in t_Y are uniquely determined by the ways in which the variables in t_X are instantiated, either because they appear in t_X directly, or because they can be inferred from variables in t_X using the dependencies induced by P.
- *Consistency*: Given a second `instance` Q `=>` $C\ s$ `where ...`, and a dependency $(X \leadsto Y) \in F_C$, then we must ensure that $t_Y = s_Y$ whenever $t_X = s_X$. In fact, on the (reasonable) 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: if t_X and s_X have a most general unifier U, then $Ut_Y = Us_Y$.

These two properties, taken from Jones [18] where they appeared without identifying names, are sufficient to guarantee that all of the declared instances and dependencies are compatible. The rules are not strictly necessary to ensure compatibility because they do not take full account of the contexts provided by P and Q. More specifically, if we can be sure that P is not satisfiable in the covering condition, or that $UP \cup UQ$ is not satisfiable in the consistency condition, then we do not need to enforce covering or consistency, respectively, to ensure compatibility with declared dependencies. However, given the open nature of Haskell type classes (i.e., the ability to extend existing classes using modules that provide new instances), it is impossible to ensure that a set of predicates will remain unsatisfiable. For practical purposes, therefore, we will consider covering and consistency as both necessary and sufficient.

The original statement of covering [18] required only $TV(t_Y) \subseteq TV(t_X)$. That is weaker than the version presented here, which allows the right hand side to be enlarged by taking its closure with respect to dependencies induced by P. Soon after the original paper was written, we realized that its formulation of covering was too weak to support some interesting applications, and the generalized version shown here was developed and implemented in Hugs (with credit to Jeff Lewis). These conditions were also studied and generalized independently by Sulzmann et al. [27], who reached the same ultimate conclusion. We have adopted their labels for these two rules, but with one subtle change. Specifically, we have renamed their 'coverage' condition as 'covering'; the intent of this rule is to show that all 'dependent' variables are 'covered' by 'determining' variables, so the 'cover' prefix is appropriate, but it has little to do with standard notions of 'coverage'.

2.5.2 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:

- Suppose that we have two predicates $C\ t$ and $C\ s$. If $t_X = s_X$, then t_Y and s_Y must be equal.
- Suppose that we have an inferred predicate $C\ t$, and an instance `instance ... => ` $C\ t'$ `where ....` If $t_X = St'_X$, for some substitution S (which can 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. If unification fails, then we have detected and can report a type error. Otherwise we have obtained an improving substitution (Section B.4). During type inference, we can apply these two rules repeatedly until no further opportunities for improvement are found, appealing to Theorems 1 and 2 in Appendix B for guarantees of completeness and soundness, respectively.

2.5.3 Detecting Ambiguity

The type checker must reject any program with an ambiguous principal type because of potential for semantic ambiguity [3, 15]. The standard definition [22, Section 4.3.4] labels a type $P \Rightarrow \tau$ as ambiguous if there is a variable $a \in TV(P)$ that is not also in $TV(\tau)$; the intuition here is that, if there is no reference to a in the body of the type, then there will be no way to determine how it should be instantiated. More relaxed notions of ambiguity can be used, however, in situations where the potentially ambiguous variable a might be resolved by some other means [15, Section 5.8.3]. So, in fact, we need not insist that every $a \in TV(P)$ is mentioned explicitly in τ, so long as it is uniquely determined by the variables in $TV(\tau)$. More precisely, if we are working with functional dependencies, then the type $P \Rightarrow \tau$ is *ambiguous* if, and only if $TV(P) \not\subseteq (TV(\tau))^+_{F_P}$.

2.6 Beyond Functional Dependencies

By inspecting the table for the `Plus` class in Section 2.3, it is not hard to see that there are other properties of the entries that cannot be captured by functional dependencies. For example:

- If `Plus a a b`, then `a`=`b`. This translates to an improvement rule (See Section B.4): [`b`/`a`] *improves* `Plus a a b`.
- If `Plus a b Int`, then `a`=`b`=`Int`. This translates to an improvement rule: [`Int`/`a`, `Int`/`a`] *improves* `Plus a b Int`.

As can be seen, however, both of these properties can be described in terms of improvement. These examples demonstrate clearly that there is potential to go beyond what functional dependencies can offer, even if we remain within the framework of improvement for qualified types. To see how these rules might be applied in practice, consider the following pair of functions, each of which provides a method for doubling an input argument:

```
double1 x = plus x x
double2 x = [0..] !! plus x x
```

For `double1`, we can initially infer `(Plus a a b) => a -> b`, and then improve that to obtain `(Plus a a a) => a -> a` using the first rule above. For `double2`, we can go further (knowing that the list indexing operation, `!!`, expects an `Int` as its right argument, and that `[0..]` produces a list of type `(Num a) => [a]`) and use the second rule to infer `double2 :: (Num a) => Int -> a`.

In fact, the original proposal for functional dependencies in Haskell was conceived as a generalization of the work on *parametric type classes* by Chen et al. [5]. Functional dependencies were never intended to serve as an ultimate mechanism for improving qualified types, but were just one 'sweet spot', hitting a nice compromise between expressive power and tractability, while also leveraging results from database theory. The examples described above show that it is possible, and perhaps even useful to go beyond what can be obtained with functional dependencies, but also raise questions about how these alternative forms of improvement might be expressed in source code. One option might be to explore the possibility of introducing general, user defined improvement rules along the lines suggested, but not subsequently developed by Jones [17]. With this approach, we would extend the language with a new declaration form, `improve` P `using` E, where P is a set of predicates and E is a list of equalities. For example, the first of the two rules above could be written as `improve (Plus a a b) using a = b`. The biggest challenge here is likely to be in figuring out general, algorithmic techniques to check for compatibility between sets of declared instances and improvement rules. Another—perhaps less expressive but also more tractable—possibility might be to extend the syntax for annotating class declarations to capture other forms of improvement beyond functional dependencies. Adopting a hypothetical syntax for 'implication annotations', we can give a revised definition for `Plus` that captures all of the properties described previously:

```
class Plus a b c | (a b -> c),
                   a=b   => b=c,
                   c=Int => a=Int,
                   c=Int => b=Int where ...
```

This idea came out of an email correspondence with Martin Sulzmann and Simon Peyton Jones. Indeed it seems that their framework for *constraint handling rules* [27] may be a good tool to help explore more advanced applications for improvement like these.

3. Functional Notation

One aspect of functional dependencies in Haskell that has perhaps caused more difficulty than any other is the apparent reliance on a *relational* notation for predicates, which some have even suggested is inappropriate for a *functional* programming language. Less subjectively, relational notation can lead to long and obfuscated types. In their work on bitdata, for example, Diatchki et al. [12] proposed the following operator for concatenating two bit strings:

```
(#) :: Add m n p => Bit m -> Bit n -> Bit p
```

Here, `m`, `n`, and `p` range over natural numbers (technically, over types of kind `Nat`); `Bit m` is the type of bit strings with precisely `m` bits; and `Add` is a three parameter class that is defined so that `Add m n p` holds if, and only if `m`+`n`=`p`. In particular, an `Add a b c` predicate has a functional dependency {`a`, `b`} $\rightsquigarrow$ {`c`}. Already, this notation is awkward because it is not possible to make sense of the final result type, `Bit p`, without also considering the predicate `Add m n p`. The situation does not improve for functions like `(\x y z -> x # (y # z))`, with most general type:

```
(Add m q r, Add n p q)
       => Bit m -> Bit n -> Bit p -> Bit r
```

To understand this type, it is necessary to 'reverse engineer' the constraint set. Once we recognize that q is simply a name for n+p, and that r is, in turn, a name for the sum m+q and hence m+n+p, then the type becomes reasonably clear. These relationships, however, are not immediately obvious from a quick glance at the type.

It is not hard to argue that there is a better name for n+p than an arbitrarily selected, fresh variable name! Because of the functional dependency on Add, we know that q is uniquely determined as a function of n and p, so it makes good sense to adopt a correspondingly functional notation. Taking a lead from the name of the class, we will write Add n p for the unique type q such that Add n p q (we will return shortly to consider the possibility that there might, in fact, be no such q). Note that there is no problem in using the same name, Add, for both a type and a predicate: The definition of Haskell already places type constructors and type classes in the same namespace [22, Section 1.4], and we can distinguish between the two uses of Add from the context in which they appear: In a class constraint, Add is a three place predicate; in a type expression, it is an arity two 'function' on types. Using this notation, we can write the type of the (#) operator more succinctly as:

```
(#) :: Bit m -> Bit n -> Bit (Add m n)
```

and the type of (\x y z -> x # (y # z)) as:

```
Bit m -> Bit n -> Bit p -> Bit (Add m (Add n p))
```

These types are shorter, and easier to understand than the originals, and provide a strong argument for adopting a functional notation.

3.1 Lightweight Functional Notation

Motivated by examples like these, Schrijvers et al. [26] have proposed an extension of Haskell that adds new syntax and type system machinery for dealing with open type-level functions. However, as observed by Diatchki and Jones [11] and subsequently elaborated and implemented by Diatchki [10, Chapter 5], it is possible to obtain essentially the same functionality without significant additions to a type system that supports functional dependencies—or, more generally, improvement, as in Appendix B—using only a lightweight syntactic abbreviation. The general case is that, if C $a_1 \ldots a_n$ is an n parameter class and the last parameter, a_n, is functionally dependent on (a subset of) the first $n-1$ parameters, then we will allow the use of C $t_1 \ldots t_{n-1}$ in type expressions as a notation for the unique type t such that C $t_1 \ldots t_{n-1}$ t. In practice, this can be implemented by replacing each occurrence of C $t_1 \ldots t_{n-1}$ in a type with a freshly generated type variable a of the appropriate kind, and then adding a new constraint, C $t_1 \ldots t_{n-1}$ a to the context. This, of course, is exactly what programmers were forced to do *by hand* in the original system, but now an implementation can handle the translation automatically instead. It is important to realize that, so far as the underlying type system is concerned, this is just a matter of how types are presented to the user, and not a substantive change. A programmer may write the type of (#) in the abbreviated form, and the same notation can be used in types reported to the user, for example, in type error messages. This should provide a friendlier notation for programming with functional dependencies in many applications. However, for the purposes of type inference or checking, types are still handled exactly as if they were written in the underlying, relational form.

To illustrate how this works in practice, we repeat the definition of the Elem class from Section 2.3 using the functional notation:

```
class Elem c e | c -> e where
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
```

The first line of this declaration still mentions the two class parameters, but the remaining lines use the expression, Elem c instead of the element type e. If the reader was wondering previously why we chose to call this class Elem instead of, say, Collects, then the reason should now be clear: using the functional notation Elem c is a way of writing the element type of the collection c. For this particular declaration, the functional notation results in slightly longer (but, arguably, easier to read) types for each of the members. A stronger case can be made for the insert2 function from Section 2.4 whose type can now be rewritten as follows:

```
insert2 :: Elem c -> Elem c -> c -> c
```

Expanding the notation by replacing each occurrence of Elem c with a fresh type variable we obtain the following type:

```
insert2 :: (Elem c a, Elem c b) => a -> b -> c -> c
```

At first glance, this seems wrong because it allows the two arguments to have different types. However, because of the functional dependency, we can quickly deduce that a=b, and then conclude that the expanded type is equivalent to the original, qualified form:

```
insert2 :: (Elem c e) => e -> e -> c -> c
```

3.2 Comparison with Other Proposals

It is instructive to see how the definition of Elem might look with other proposals. Using an associated type [4], for example, a suitable declaration might be as follows:

```
class Collects c where
  type Elem c
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
```

This syntax trades a functional dependency annotation for an inner declaration of the Elem type, but is otherwise very similar to the version in the previous section. One difference is that it introduces two new names, Collects and Elem, where our version uses only one. While there is some economy in using only one name, there may occasionally be reasons to prefer two. We can easily handle such cases using an extra class:

```
class Collects c where
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
class Collects c => Elem c e | c -> e
```

In the associated types proposal, each instance of Collects must provide a corresponding definition for the Elem type. The same applies for the version that we have just given using two classes except that the definition of an Elem associated type will be replaced by the definition of an instance of the Elem class.

Using open type functions [26], we again have two names, but this time Elem denotes a type family (i.e., an open, or extensible mapping from collection types to element types):

```
type family Elem c
class Collects c where
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
```

Again, there is only a small syntactic delta from the definition of Elem using functional dependencies.

As these examples suggest, the different proposals are notationally very similar, although there are some cases where associated types or type functions require the introduction of more names than are needed with functional dependencies. Apart from minor differences in syntax, however, we believe that these approaches are essentially interchangeable, and that they have the same expressive power. One disadvantage of associated types and type functions is that they introduce new language mechanisms whose interaction with the rest of the type system—including type classes and higher-kinded polymorphism—must then be explored and documented. The framework of functional dependencies, by comparison, avoids this because it is already fully integrated with the type system.

3.3 When Relational Notation Cannot Be Avoided

Although appealing, it is important to note that there are some cases where functional notation, by itself, is not enough. Consider, for example, the `insx` function, as described by Schrijvers et al. [26], that inserts an `'x'` character into a collection, `c`:

```
insx  :: (Elem c Char) => c -> c
insx c = insert 'x' c
```

There is no way to avoid a relational constraint here because the collection type is not a function of the element type. Dealing with this example using type functions is more involved and led Schrijvers et al. [26] to introduce a generic, type equality predicate of the form `t1 ~ t2` so that they can give the following type for `insx`:

```
insx  :: (Collects c, Elem c ~ Char) => c -> c
```

This approach, however, has two problems. First, it results in a type that has two predicates where only one is really needed; this is a result of having defined the `Elem` type family without reference to the `Collects` class. Second, it leads the authors into a much more complicated framework for solving entailment problems because it has to deal with all the challenges of automated reasoning with a reflexive, symmetric, transitive equality operator. For example, it becomes necessary to consider how constraints of the form `F a ~ G (F a)` should be handled, where both `F` and `G` are unary type functions. The problem here, of course, is that a naive unfolding of the equality could lead to non-termination by rewriting the type `F a` to `G (F a)` and then to `G (G (F a))`, and so on.

Functional dependencies avoid these kinds of problem by adopting a much simpler language of constraints. There is no need, for example, to introduce equality constraints because we already have the necessary relational notation to fall back on when it is needed. With functional dependencies, the `F a ~ G (F a)` constraint becomes `F a (G (F a))`, which then simplifies, by expanding the abbreviation for `G` with a new variable `c`, to `(F a c, G (F a) c)`, and then, by a further expansion, to `(F a c, G d c, F a d)`. Finally, using the assumed dependency for `F`, we can conclude that `c`=`d`, and hence obtain the final result `(F a c, G c c)`. No further expansion is possible at this point, and so the process terminates without further ado.

Although they may be useful for other applications, we do not believe that fully general equality constraints are needed to support functional notation. Taking a lead from the work on type functions, however, it might be worth allowing the notation $\mathtt{C}\ \mathtt{t}_1 \ldots \mathtt{t}_{n-1}$ `~ a` to be used as special syntax for an n-parameter class in which the last parameter is functionally dependent on the first $(n-1)$ parameters. With this notation, we can rewrite the definition of `Elem` one more time with the underlying functional dependency being implied by the use of the `~` symbol: `class Elem c ~ e where ....` It should be strongly emphasized, however, that this notation does not provide a general equality constraint. The constraint `Elem c ~ e` is just another way of writing `Elem c e`, and we cannot even assume, for example, that the expression `e ~ Elem c` is syntactically well-formed.

3.4 Caveats for Functional Notation

Up to this point, Haskell programmers have always been able to make sense of type expressions like `T a` as the application of a type constructor, `T`, to a parameter, `a`, even if they have never seen the definition of `T`. This will change, however, if we adopt any one of the proposals for functional notation discussed previously because now programmers must be prepared to deal with the additional possibility that `T` is actually an associated type, a type family, or a type class. Moreover, use of these functional notations may conceal the use of overloading, and of associated partiality. For example, we can write the type of `insert` as `Elem c -> c -> c`, which, if we assume the original conventions of Haskell, looks like a fully polymorphic type in which `c` can be instantiated to any type. The truth, of course, is that this type will only make sense for certain choices of `c`, and that the function is actually overloaded (which, if forgotten, could result in some puzzling error messages courtesy of Haskell's *monomorphism restriction*). In this respect, the original, relational version of the type, `(Elem c e) => e -> c -> c`, while more verbose, is also more honest because it reflects the behavior of `insert` more directly. Although few will regard these issues as show-stoppers, it is important to recognize that none of these proposals for functional notation comes without a cost.

3.5 Type Synonyms as a Special Case

Type synonyms are a special kind of type function that have been supported in Haskell since the first versions of the language. A typical type synonym declaration takes the form:

$$\mathtt{type\ T}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n\ =\ \mathtt{t}.$$

The intention here is that a type expression of the form $\mathtt{T}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n$ is just an abbreviation for the corresponding substitution instance, $[\mathtt{t}_1/\mathtt{a}_1,\ \ldots\ \mathtt{t}_n/\mathtt{a}_n]\mathtt{t}$, of the right hand side type, `t`. Although type synonyms are simple and natural to use, they can be quite awkward to implement because of the tension between ease of type checking and quality of error messages: It would be easy to handle type synonyms if they were fully expanded before type checking, but then any types appearing in error messages would also be expanded, which might therefore be harder for programmers to understand.

Perhaps unsurprisingly, we can view type synonyms as a special case of the functional notation described here. The example above, for example, corresponds to a pair of declarations, one that defines `T` as an $(n+1)$-parameter class:

$$\mathtt{class\ T}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n\ \mathtt{a}\ \mid\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n\ \texttt{->}\ \mathtt{a},$$

and another that defines a single instance of this class:

$$\mathtt{instance\ T}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n\ \mathtt{t}.$$

Although this is unlikely to simplify the task of implementing type synonyms in a significant way, it is helpful to see that there is potential for sharing implementation costs (and conceptual understanding) for functional dependencies with those for type synonyms.

3.6 Functional Notation without Dependencies

The technique that we have been using to support functional notation has two components: (1) replacing a 'partially applied' predicate in a type with a fresh type variable and a fully applied predicate in the associated context; and (2) using functional dependencies to improve the resulting type. In fact, it is possible to decouple these two pieces, and to allow partially applied predicates in type expressions even if there is no associated dependency. Without practical experience, it is hard to know whether this will turn out to be a good idea, or a step too far that will only confuse unsuspecting newcomers. Nevertheless, we believe that it is interesting enough to document this possibility here for future consideration.

The basic idea is to allow an expressions like $\mathtt{C}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_{n-1}$ to be used in type expressions where `C` is any n-parameter type class. Each such expression can be replaced with a fresh type variable, `a`, so long as we also add a new predicate, $\mathtt{C}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_{n-1}\ \mathtt{a}$ to the associated context. The difference from what we have described previously is that we will allow this notation to be used *even if there are no dependencies for the class* `C`. In particular, this includes standard classes like Eq, Show, and Monad. The following list shows how the types of some standard Haskell operators appear when written, more concisely, in this notation:

```
show           :: Show -> String
fromIntegral   :: Integral -> Num
fromInteger    :: Integer  -> Num
ceiling        :: RealFrac -> Integral
properFraction :: RealFrac a => a -> (Integral, a)
return         :: a -> Monad a
lift           :: Monad m => m a -> MonadTrans m a
pi             :: Floating
```

One problem with this notation is that it may be hard for programmers to understand types like `Integral -> Bool` because the range and domain types look so similar, at least from a purely syntactic perspective. To interpret this type fully, a reader has to know that `Integral` is a class while `Bool` is a regular type. And, of course, there are many places where this notation cannot be used. For example the type of the equality operator, `Eq a => a -> a -> Bool` cannot be abbreviated to `Eq -> Eq -> Bool`, which, instead, abbreviates a different type, `(Eq a, Eq b) => a -> b -> Bool)`. This notation is also not applicable in types where a single variable is subject to multiple class constraints, such as `(C a, D a) => a -> a`.

4. Miscellaneous Further Issues

This section provides a brief summary of the technical problems with the original proposal/implementation of functional dependencies that are not already discussed elsewhere in this paper.

Type Checking. Although the implementation of functional dependencies in Hugs was based on the type system of SIQT (See Appendix B), some parts of the implementation were not properly updated when it was modified to support functional dependencies. For example, the following code is not accepted by Hugs:

```
class C a b | a->b where ...
instance C Int Bool where ...

f  :: C Int a => a -> a
f x = x && True
```

According to the theory of SIQT, the declared type for `f` that is shown here is perfectly valid, and equivalent to the inferred type `Bool -> Bool`. The Hugs implementation, however, uses an older, purely syntactic algorithm to determine equivalence of declared and inferred types, without allowing for the possibility of improvement, and hence rejects the declared type for `f`. Although it doesn't really matter in this case (because a programmer can substitute the more specific type for `f` in the source), there are some situations—for example, in the body of an instance declaration—where this behavior can cause the type checker to reject valid code. We consider this to be a bug in Hugs that should be fixed to ensure compliance with the parts of SIQT theory that it is supposed to implement! (GHC, by the way, exhibits similar behavior.)

Implied Dependencies. Haskell allows classes to be defined in a hierarchy, but the design that is used for functional dependency annotations does not reflect this. The following code, for example, defines a class called `C`, and a subclass of `C` called `D`.

```
class C a b | a->b where ...
class C a b => D a b where ...
```

The class `C` shown here carries a functional dependency, and it is not actually possible to define an instance of `D` that does not also satisfy the same dependency. However, nothing in the declaration of `D` reflects the fact, other than implicitly through the use of `C` as a superclass. We consider this to be a design error, and believe that programs will be easier to understand if every class is annotated with dependencies that are at least as strong as those implied by its superclass. For the particular example shown here, this would mean that the definition of `D` should be written as:

```
class C a b => D a b | a -> b where ...
```

Decidability and Termination. Although it is not strictly required, it is certainly desirable to ensure that the process of type checking will terminate (that is, without the need to set some arbitrary bound on complexity within the type checker, and without relying on user intervention). This can often be accomplished by imposing restrictions on the syntactic form of class and instance declarations, albeit at the cost of limiting expressiveness. We do not address this topic further here, but note that it has already received careful attention from others, including Sulzmann et al. [27].

Interaction with Other Features. Implementations of Haskell often support a range of experimental features and extensions to the type system such as overlapping instances, higher-rank polymorphism, existential types, extensible records, and GADTS. Unfortunately, the interactions of these features with functional dependencies have not, to the best of our knowledge, been formally studied. Although we do not have any particular reason to expect difficulties, some due diligence is required to work through the details.

5. Program Design with Functional Dependencies

When functional dependencies were first introduced, we paid little attention to explaining how they should be used in program design. In database systems, functional dependencies are an important tool for working with large tables because they make it possible to automate aspects of analysis and schema design. We did not expect that it would be necessary to spend time on such issues for type classes where there are typically many fewer columns, and where programmers would more naturally gravitate to designs. With hindsight, of course, these were not realistic expectations, and we now recognize that there is much to be gained from articulating and sharing principles for good design. In this section, we begin that process, albeit briefly, by considering the use of dependencies in two specific pieces of widely used Haskell code, both of which are available for download from `http://hackage.haskell.org`.

Dependencies not required. Version 1.1.0.0 of MTL, the Haskell Monad Transformer Library, uses multiple parameter type classes to categorize different families of monad. The `MonadState` class, for example, that was discussed in Section 2.2, is one such example, except that the version in MTL attaches a dependency from the monad type to the state type. This is unfortunate because it limits the applicability of `MonadState` to monads with only a single state component. It is also ironic because, as described previously, examples like `MonadState` were among the few early applications of multiple parameter type classes that worked well, without any need for additional type annotations [20, 23]!

Using normal forms. Version 3.0.0 of the Parsec library has been generalized to allow parsing over multiple monad and token types, with the following `Stream` class playing a central role:

```
class Monad m => Stream s m t | s -> t where
  uncons :: s -> m (Maybe (t,s))
```

From a database perspective, however, the dependency leaves `m` completely unconstrained, and we can conclude that `Stream` is not in second normal form (A.4). The types of some of the most commonly used Parsec combinators are also a little unusual. For example, the following type signature includes a constraint `Stream s m t`, but the variable `t` is not used in the main type:

```
many :: Stream s m t
          => ParsecT s u m a -> ParsecT s u m [a]
```

There is no ambiguity here—`t` is uniquely determined by `s`—but we might still interpret this an indicator of potential problems. In fact, we can obtain second normal form by decomposing `Stream` into two separate pieces, as shown by the following pair of class declarations (and directly reflecting the example in Section A.4):

```
class (Monad m) => Stream s m
class Tok s t | s -> t where
  uncons :: (Stream s m) => s -> m (Maybe (t,s))
```

We have used this observation to guide a refactoring of the Parsec codebase to use the two classes shown here in place of the original three parameter version of `Stream`. The refactoring goes through smoothly, and results in simplified types for many of the exported combinators, including the `many` function mentioned previously, where the original `Stream s m t` is replaced with `Stream s m`, appropriately avoiding any mention of `t`. Although a programmer may, perhaps, have reasons to adopt a design that is not in normal form, in this case it appears to result in a program that is easier to understand (because the types are simpler) and easier to extend (because we can add instances to either `Stream` or `Token` without being required to add instances to both). This suggests that designers of classes with dependencies may be able to simplify and improve their code by taking account of normalization.

6. Conclusions

The original proposal for functional dependencies provided Haskell programmers with a tool that they could use to work more effectively with multiple parameter type classes. It also provided a boost to the exploration of programming techniques that rely on a notion of computation at the level of types. Some aspects, however, have proved to be difficult to understand, or awkward to apply. The updated design that we have described in this paper addresses those problems in two key ways, first by emphasizing the use of a simple functional notation that provides a more comfortable syntax (Section 3), and second by providing clearer guidelines on the use (and potential misuse) of functional dependencies (Section 5). Overall, the technical changes to the original proposal are relatively minor, but we believe that they will have a significant impact on usability.

As for the debate over whether the next version of Haskell should adopt functional dependencies or type functions, we obviously believe that there are very good reasons to adopt the former, including the direct integration with type classes, and the lightweight treatment of functional notation that it provides. That said, if the selection is made carefully, on the basis of accurate information, and with benefit to the entire Haskell community as the primary consideration, then we will happily accept any outcome!

Appendices

A. Functional Dependencies in Database Theory

Our treatment of functional dependencies in Haskell is based directly on ideas that were originally introduced many years earlier in the study of relational databases [1, 6], and that are well-documented in standard textbooks [2, 24]. Our experience, however, is that Haskell developers who are using functional dependencies are not always familiar with that material. This appendix is intended to fill that gap by providing a summary of functional dependencies in the context of relational databases.

A.1 Relations and Relational Databases

A relational database can be described by a family of relations, each of which may be drawn as a table with zero or more rows. The following table, for example, captures the results of a hypothetical survey of a company's employees about how they get to work, and thus corresponds to a relational database with only one table.

SURVEY

Employee	Residence	Transport
Alice	Harbortown	Car
Alice	Harbortown	Walk
Bob	Hillville	Bike
Bob	Hillville	Bus
Bob	Hillville	Car
Carol	Hubford	Bus
David	Hubford	Train

This table has three *columns* and seven *rows*. Individual rows may be described by tuples of the form (e, r, t), where e identifies an employee, r is the town where they live, and t is a method of transport. More generally, a database may be described by a *schema* that gives a name for each table as well as a name (and perhaps also a type) for each column. From a Haskell programmer's perspective, a schema is like a type that describes a set of possible databases.

More formally, a *table* T is a relation over an indexed family of sets $\{D_i\}_{i \in I}$, where I is a set of index values (i.e., column headings) and D_i is the type of values in column i. Our SURVEY table, for example, can be viewed as a relation indexed by the set $I = \{\text{Employee, Residence, Transport}\}$. The elements of such a table are *tuples*, each of which is an indexed family of values $\{t_i\}_{i \in I}$ such that $t_i \in D_i$ for each $i \in I$. Note that, if $I = \{1, \ldots, n\}$, then 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 $i \in I$, then we write t_i for the i^{th} component of t. Similarly, if $X \subseteq I$, then we write t_X, pronounced "t at X", for the projection of the tuple t onto the columns in X. Intuitively, t_X just picks out the values of t for the indices appearing in X, and discards any remaining components.

A.2 Functional Dependencies

A database schema will typically impose certain *integrity constraints* to characterize the permitted structure of individual database tables more precisely. In the SURVEY table, for example, it is reasonable to assume that each employee has only one home town. We can capture this as a *functional dependency*, written $\{\text{Employee}\} \rightsquigarrow \{\text{Residence}\}$, which asserts that, if two rows in SURVEY have the same Employee, then they also have the same Residence. On the other hand, it is clear that there is no such dependency between the Employee and Transport columns because some employees use more than one method of transport. Adding dependencies to a schema restricts the ways in which tables can be populated, and a good database management system will ensure that the dependencies are maintained as the tables are extended or updated. For example, if David moves to Hillville, then we could *replace* the tuple $t_1 = (\text{David,Hubford,Train})$ in SURVEY with a new tuple $t_2 = (\text{David,Hillville,Train})$, but we should not just add t_2 to the existing table because that would violate the dependency.

A particular table may sometimes satisfy 'accidental' dependencies that we would not wish to include in the schema. As it happens, for example, the combination of Residence and Transport is enough to determine a unique Employee in the given SURVEY table. This observation can be captured by the dependency $\{\text{Residence,Transport}\} \rightsquigarrow \{\text{Employee}\}$. It is unlikely, however, that we would want to assume or enforce this dependency because that would prevent us from including rows for distinct employees that happen to have the same Residence and Transport. It is the job of the database designer, using their knowledge of the application domain, to identify the set of functional dependencies that should be associated with a given table.

A.3 Formalizing Functional Dependencies

Formally, if T is a table indexed by a set I, then a *functional dependency* is a pair of the form $X \leadsto Y$, read as "X determines Y," where X and Y are both subsets of I. If X and Y are known sets of elements, say $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, then we will often write the dependency $X \leadsto Y$ in the form $x_1 \ \ldots \ x_n \leadsto y_1 \ \ldots \ y_m$. If a table T satisfies a dependency $X \leadsto Y$, then the values of any tuple at Y are uniquely determined by the values of that tuple at X. We formalize this as follows (pronouncing $\models$ as "satisfies"):

$$T \models X \leadsto Y \iff \forall t, s \in T.(t_X = s_X) \Rightarrow (t_Y = s_Y).$$

This also extends to sets of dependencies:

$$T \models F \iff \forall (X \leadsto Y) \in F. T \models X \leadsto Y.$$

For example, if we take $I = \{1, 2\}$, then the tables satisfying $\{\{1\} \leadsto \{2\}\}$ are just the partial functions from D_1 to D_2, and the tables satisfying $\{\{1\} \leadsto \{2\}, \{2\} \leadsto \{1\}\}$ are the partial injective functions from D_1 to D_2.

It is also possible to reason about functional dependencies using inference rules for *reflexivity*, *transitivity*, and *augmentation*:

$$\frac{X \supseteq Y}{X \leadsto Y} \qquad \frac{X \leadsto Y \quad Y \leadsto Z}{X \leadsto Z} \qquad \frac{X \leadsto Y}{X \cup Z \leadsto Y \cup Z}$$

These are sometimes referred to as *Armstrong's Axioms* after William Armstrong [1], who showed that they are correct and complete. These rules can also be extended to sets of dependencies; we will write $F_1 \vdash F_2$ if all of the dependencies in F_2 can be deduced from the dependencies in F_1. If we have both $F_1 \vdash F_2$ and $F_2 \vdash F_1$, then the two sets of functional dependencies are equivalent, and we refer to either one as a *cover* for the other. It is easy to find algorithms for computing minimal/optimal covers for sets of functional dependencies in standard textbooks on the theory of relational databases [2, 24]. Such covers are of practical interest because they provide the most concise possible characterization of a set of dependencies.

The *closure*, J_F^+, of a set $J \subseteq I$ with respect to a set of functional dependencies F is another useful, textbook concept from the theory of relational databases, and is the smallest set such that:

- $J \subseteq J_F^+$; and
- If $(X \leadsto Y) \in F$, and $X \subseteq J_F^+$, then $Y \subseteq J_F^+$.

For example, if $I = \{1, 2\}$, and $F = \{\{1\} \leadsto \{2\}\}$, then $\{1\}_F^+ = I$, and $\{2\}_F^+ = \{2\}$. Intuitively, the closure J_F^+ is just the set of indices that are uniquely determined, either directly or indirectly, by the indices in J and the dependencies in F. Closures are easy to compute using a simple, fixed point iteration.

A.4 Database Normalization

Although the SURVEY table in Section A.1 may be sufficient for the original application, it also suffers from some structural problems. For example, the table duplicates Residence information for Alice and Bob. As a result, if we want to change the Residence for Bob, then we must update three distinct rows to ensure that we do not violate the $\{\text{Employee}\} \leadsto \{\text{Residence}\}$ dependency. Another problem is that we cannot add new tuples unless we have information for all three columns. If a new employee, Elly from the town of Hillville, joins the company, then we cannot add information for her to the table until we have also determined what methods of transportation she will be using.

A range of techniques—referred to as *database normalization*—have been developed to guide the design of relational database tables that avoid problems like these. The fact that a given database is not in normal form is not an absolute indicator of bad design, but it is a strong hint that a better design might be possible. The specific problems with SURVEY, for example, are a symptom of the fact that the table is not in (second) normal form. In this case, however, it is possible to decompose SURVEY into two distinct tables, HOMETOWN and COMMUTES, without loss of information:

HOMETOWN

Employee	Residence
Alice	Harbortown
Bob	Hillville
Carol	Hubford
David	Hubford

COMMUTES

Employee	Transport
Alice	Car
Alice	Walk
Bob	Bike
Bob	Bus
Bob	Car
Carol	Bus
David	Train

$$\begin{array}{ll} (e, r, t) \in \text{SURVEY} & \\ \iff & (e, r) \in \text{HOMETOWN} \\ \wedge & (e, t) \in \text{COMMUTES} \end{array}$$

The formula here expresses the relationship between the original and the normalized versions of the database. In the terminology of relational databases, it expresses SURVEY as the *relational join* of HOMETOWN and COMMUTES. It is clear that the normalized version avoids duplication of Residence information because it collects that data independently from Transport values in the HOMETOWN table, which includes just one row for each Employee. In addition, it is possible to add a tuple (Elly,Hillville) to HOMETOWN without requiring any information about how Elly travels to work.

The first, second, and third normal forms were introduced by Edgar Codd [6, 7, 8]; several higher normal forms have been introduced subsequently. To be in first normal form (1NF), a table should correspond directly to a relation; for example, it should not include duplicate rows. It is also common to require that the values in each tuple are, in some sense, 'atomic', suggesting that set-valued columns and nested tuples, for example, should not be allowed. All of the tables that we consider in this paper are trivially assumed to be in 1NF.

To define the second and third normal forms more precisely, it is helpful to introduce some additional terminology. If T is a table indexed by I with a set of functional dependencies F, then we say that a set $K \subseteq I$ is a *superkey* if $K_F^+ = I$; in other words, K is a superkey if the fields in K are sufficient to identify tuples in T uniquely. If, in addition, K is minimal (i.e., no proper subset of K is a superkey), then we say that it is a *key* for T. An index $i \in I$ that is not part of any key is said to be a *non-prime attribute* of T. The keys and non-prime attributes for each of the tables that we have used here are as follows (in this particular group of examples, all of the tables have a unique key; it is not difficult, however, to construct tables that have multiple distinct keys):

Table	Key	Non-prime attributes
SURVEY	{Employee, Transport}	Residence
HOMETOWN	{Employee}	Residence
COMMUTES	{Employee, Transport}	-

To be in second normal form (2NF), a table must be in 1NF and every non-prime attribute must be functionally dependent on every key, but not on any proper subset of a key. The HOMETOWN table satisfies 2NF because Residence is uniquely determined by Employee, and the COMMUTES table trivially satisfies 2NF because there are no no-prime attributes. On the other hand, SURVEY is not 2NF because the non-prime attribute Residence is functionally dependent on a subset of the key.

To be in third normal form (3NF), a table must be in 2NF and every non-prime attribute must be non-transitively dependent on each key. The latter condition means that, if K is a key, and i is a non-prime such that $K \leadsto J$ and $J \leadsto \{i\}$, then either J is a key or else $i \in K \cup J$. The SURVEY table does satisfy this latter property, failing to be in 3NF only because it is not in 2NF. As a simple example that is not in 3NF, consider a table indexed by columns {Person, Hometown, Country} and dependencies {Person} $\rightsquigarrow$ {Hometown} (every person has a unique home town) and {Hometown} $\rightsquigarrow$ {Country} (every town is in a single country). Once again, we could achieve 3NF for this particular database by decomposing the three column table into two separate tables, one for each dependency.

B. Improving Qualified Types

This section contains a summary of the concept of *improvement* of the kind introduced in "Simplifying and Improving Qualified Types" [16], hereafter referred to as SIQT. This provides the theoretical foundation for our work on functional dependencies. In fact SIQT was developed as a general framework of type systems with constrained polymorphism, covering not only Haskell type classes, but also subtyping, record types, and other applications. Because the main topic of this paper has to do with type classes and there are many examples of that in the body of the paper, we focus here on an example from a slightly different area—subtyping—so as to emphasize the general nature of SIQT.

B.1 Interpreting Type Schemes

In a type system with parametric polymorphism, the assignment of a *type scheme* $\forall a.a \to a$ to the identity function $id = \lambda x.x$ is an indication that *id* may be used with any of the types in the set:

$$[\![\forall a.a \to a]\!] = \{\, \tau \to \tau \mid \tau \in \mathit{Type} \,\}.$$

More generally, of course, we define:

$$[\![\forall a_1.\ldots\forall a_n.\tau]\!] = \{\, S\tau \mid \mathit{dom}\, S \subseteq \{a_1, \ldots, a_n\} \,\},$$

where S ranges over substitutions of types for type variables, and $\mathit{dom}\, S$ is the set of type variables a for which $Sa \neq a$. Note that this interpretation induces an ordering between type schemes in which $\sigma \geq \sigma'$ if, and only if $[\![\sigma]\!] \supseteq [\![\sigma']\!]$. In fact, this gives the same ordering that Damas and Milner [9] used—specifying that σ' is a *generic instance* of σ—to demonstrate the existence of most general, or *principal types* for programs in ML.

B.2 The Theory of Qualified Types

The theory of qualified types [15] extends the syntax of type schemes to allow for the inclusion of constraints or *predicates* on types. A general type scheme in this setting takes the form $\forall a_1.\ldots\forall a_n.P \Rightarrow \tau$, where P is a set of predicates. The intent of a qualified type scheme is to restrict, or *qualify* the way in which quantified variables are instantiated. As a notational convenience, we will elide the $\forall a_1.\ldots\forall a_n$ and $P \Rightarrow$ portions of a type scheme if there are no quantified type variables or no predicates, respectively.

The structure of predicates varies from one application to another. For example, we can write $Eq\, a$ to specify that a is an equality type in Haskell, but, in other language settings, we might use $s \subseteq t$ to specify that s is a subtype of t, or $(l : p) \in r$ to indicate that r should be a record type containing a field labeled l of type p. The relationships between predicates are typically described by the definition of an *entailment relation*, usually written as an infix $\Vdash$, between sets of predicates. An entailment relation is required to satisfy three general properties, which we refer to as *monotonicity*, *transitivity*, and *closure under substitution*:

$$\frac{P \supseteq Q}{P \Vdash Q} \qquad \frac{P \Vdash Q \quad Q \Vdash R}{P \Vdash R} \qquad \frac{P \Vdash Q}{SP \Vdash SQ}$$

Beyond these, the specific properties of an entailment relation will again vary from one application to another. For example, a simple system of subtyping can be described by defining entailment as the smallest relation that is closed under the three general rules above and also includes the axioms $\emptyset \Vdash \{Int \subseteq Real\}$, $\emptyset \Vdash \{t \subseteq t\}$, and $\{t \subseteq t', s' \subseteq s\} \Vdash \{(t' \to s') \subseteq (t \to s)\}$ for all types t, t', s, and s', the latter being the standard rule for subtyping on functions.

The concept of *satisfiability* plays an important role in the following sections. In particular, we say that a set of predicates P is satisfiable if $\emptyset \Vdash SP$ for some substitution S. Equivalently, we say that P is satisfiable if $\lfloor P \rfloor \neq \emptyset$, where $\lfloor P \rfloor = \{\, SP \mid \emptyset \Vdash SP \,\}$ is the set of *satisfiable instances* of P.

B.3 Interpreting Qualified Type Schemes

If an expression E has an associated type scheme σ, then we expect to be able to use E as having any of the types in $[\![\sigma]\!]$—except that now we must generalize the definition of $[\![\sigma]\!]$ to account for predicates that appear in σ[3]:

$$[\![\forall a_1.\ldots\forall a_n.P \Rightarrow \tau]\!] = \{\, S\tau \mid \mathit{dom}\, S \subseteq \{a_1, \ldots, a_n\},\ \emptyset \Vdash SP \,\}$$

For example, if $negate :: \forall a.(a \subseteq Real) \Rightarrow a \to a$, and assuming only the rules for subtyping from Section B.2, then we would expect that *negate* can be used: (i) as a function of type $Int \to Int$ (because $\emptyset \Vdash Int \subseteq Real$); or (ii) as a function of type $Real \to Real$ (because $\emptyset \Vdash Real \subseteq Real$). These are the only possible types (i.e., the only elements of $[\![\forall a.(a \subseteq Real) \Rightarrow a \to a]\!]$). As a more extreme example, if the set of predicates in a scheme σ is not satisfiable, then $[\![\sigma]\!]$ is empty, and hence there will be no way to use a value of that type.

As before, we will use the ordering on type schemes in which $\sigma \geq \sigma'$ if, and only if $[\![\sigma]\!] \supseteq [\![\sigma']\!]$. Note that, if σ or σ' includes predicates, then this ordering depends not only on the syntactic form of σ and σ' but also on the definition of entailment. For that reason, and to distinguish it from the purely syntactic ordering of Damas and Milner, we sometimes refer to this relationship as the *satisfiability ordering* on type schemes.

B.4 Improvement

In some cases, a qualified type may suggest a greater degree of flexibility than is actually present. For example, the type scheme $\sigma_1 = \forall a.(a \subseteq Int) \Rightarrow a \to a$ appears to be polymorphic but, in fact, the set $[\![\sigma_1]\!]$ contains only one type, $Int \to Int$. It follows that σ_1 is equivalent to $\sigma_2 = Int \to Int$ with respect to the satisfiability ordering on type schemes. Nevertheless, there is a sense in which σ_2 is better than σ_1 because it gives a more 'honest' and certainly more concise characterization of the type.

To formalize this idea, SIQT introduces the idea of an *improving substitution*: a substitution S is said to improve a set of predicates P, written $S\ \mathit{improves}\ P$, if applying S to P does not change the resulting set of satisfiable instances; that is, if, and only if $\lfloor P \rfloor = \lfloor SP \rfloor$. The identity substitution is a trivial improving substitution for any set of predicates. However, it is often possible to compute more interesting improving substitutions. According to the preceding definition, for example, it follows that $[Int/a]\ \mathit{improves}\ \{a \subseteq Int\}$. To make use of improvement during type inference, SIQT extends the general algorithm for qualified types with the following rule:[4]

$$\frac{P \mid SA \overset{\mathrm{W}}{\vdash} E : \tau \qquad S'\ \mathit{improves}\ P}{S'P \mid S'SA \overset{\mathrm{W}}{\vdash} E : S'\tau}$$

[3] In the original presentation [16], $[\![\sigma]\!]$ was written as $[\![\sigma]\!]^{sat}_{P_0}$ to emphasize the use of information about predicate satisfiability, and to allow the empty set, $\emptyset$, shown here to be replaced with an arbitrary set of predicates P_0.

[4] The four components of each judgement $P \mid SA \overset{\mathrm{W}}{\vdash} E : \tau$ are a set of predicates, P, a set of typing assumptions, SA, an expression, E, and a (monomorphic) type, τ. We use the $\overset{\mathrm{W}}{\vdash}$ symbol to distinguish judgements for the type inference algorithm—a variant of Milner's algorithm W—from type system judgements written using the $\vdash$ symbol. The full set of typing rules and details of the type inference algorithm are presented in SIQT [16].

This allows for the calculation and application of improving substitutions in a non-deterministic fashion at any point during type inference. In practice, a particular implementation would use a more deterministic strategy, such as choosing to calculate improving substitutions as part of the generalization step. In fact, if an attempt to compute an improving substitution discovers that an inferred predicate set is not satisfiable, then it is actually possible for the inference algorithm to terminate immediately and report an appropriate type error. This may provide for earlier detection of type errors than is possible without improvement, and does not compromise the soundness or completeness theorems described below.

The key results of SIQT are the following pair of soundness and completeness results for the type inference algorithm with respect to underlying type system (The results are slightly simplified here for the purposes of this presentation):

THEOREM 1 (Soundness). *If $P \,|\, SA \overset{W}{\vdash} E : \tau$, then $P \,|\, SA \vdash E : \tau$.*

THEOREM 2 (Completeness). *If $\emptyset \,|\, A \vdash E : \sigma$ and $[\![\sigma]\!] \neq \emptyset$, then the type inference algorithm for E in A will not fail, and, for any P and τ such that $P \,|\, A \overset{W}{\vdash} E : \tau$, we have $Gen(A, P \Rightarrow \tau) \geq \sigma$.*

The completeness result, in particular, tells us that, if an expression E is well-typed, then the type inference algorithm will compute a *principal satisfiable type scheme*, $Gen(A, P \Rightarrow \tau)$, that is at least as general, with respect to the satisfiabilty ordering on type schemes, as any other (satisfiable) type that can be assigned to E. This is conceptually similar to the standard result for existence of principal types in ML, but it is also a little weaker because the satisfiability ordering is coarser than the Damas and Milner ordering, equating types like σ_1 and σ_2 above that are distinct in ML. (Note, however, that the orderings coincide on all unqualified types.)

To get the most benefit from SIQT, we would aim for an algorithm that calculates substitutions that give, in some sense, the best possible improvement. However, while this is *desirable*, it is not an absolute *requirement*. This is important because the definition of general predicate systems does not guarantee the existence of 'optimal' improvements, or of computable algorithms for calculating them. Instead, SIQT provides a general framework that allows type system designers to make an appropriate compromise between decidability/termination of type checking and improvement/quality of inferred types in cases where it is not possible to satisfy both goals.

References

[1] William Ward Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

[2] Paolo Atzeni and Valeria De Antonellis. *Relational Database Theory*. Benjamin/Cummings, 1993. ISBN 0-8053-0249-2.

[3] Stephen Blott. *An Approach to Overloading with Polymorphism*. PhD thesis, Department of Computing, University of Glasgow, December 1992.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 1–13, Long Beach, California, USA, January 2005.

[5] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *ACM Conference on LISP and Functional Programming*, pages 170–181, 1992.

[6] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] E. F. Codd. Normalized data base structure: A brief tutorial. *IBM Research Report, San Jose, California*, RJ935, 1971.

[8] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[9] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, NM, January 1982.

[10] Iavor S. Diatchki. *High-level Abstractions for Low-level Programming*. PhD thesis, OGI School of Science & Engineering at Oregon Health & Science University, May 2007.

[11] Iavor S. Diatchki and Mark P. Jones. Strongly typed memory areas. In *Proceedings of ACM SIGPLAN 2006 Haskell Workshop*, pages 72–83, Portland, Oregon, September 2006.

[12] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. High-level views on low-level representations. In *ICFP 2005: ACM SIGPLAN International Conference on Functional Programming*, 2005.

[13] Thomas Hallgren. Fun with functional dependencies, or (draft) types as values in static computations in Haskell. In *Proceedings of the Joint CS/CE Winter Meeting*, Varberg, Sweden, January 2001.

[14] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego, California, USA, June 2007.

[15] Mark 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.

[16] Mark P. Jones. Simplifying and improving qualified types. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995.

[17] Mark P. Jones. Simplifying and improving qualified types. Research Report YALEU/DCS/RR-1040, Yale University, New Haven, Connecticut, USA, June 1994.

[18] Mark P. Jones. Type classes with functional dependencies. In *ESOP 2000: European Symposium on Programming*, March 2000.

[19] Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.

[20] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925. Springer-Verlag LNCS, Båstad, Sweden, May 1995.

[21] Simon Peyton Jones. Indexed type families in Haskell, and death to functional dependencies (slides). In *AngloHaskell 2007*, Cambridge, England, August 2007. Available online at `http://haskell.org/haskellwiki/AngloHaskell/2007`.

[22] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[23] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[24] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0.

[25] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In *Proceedings of The 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

[26] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, Victoria, British Columbia, Canada, September 2008.

[27] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17:83–129, 2007.

[28] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 1989)*, pages 60–76, Austin, Texas, USA, January 1989.