Reasoning about Nested Functional Dependencies *

Carmem S. Hara[†]and Susan B. Davidson
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
Phone (215) 898-3490, Fax (215) 898-0587
Email: chara@saul.cis.upenn.edu, susan@central.cis.upenn.edu

Abstract

Functional dependencies add semantics to a database schema, and are useful for studying various problems, such as database design, query optimization and how dependencies are carried into a view. In the context of a nested relational model, these dependencies can be extended by using path expressions instead of attribute names, resulting in a class of dependencies that we call *nested functional dependencies* (NFDs). NFDs define a natural class of dependencies in complex data structures; in particular they allow the specification of many useful intra- and inter-set dependencies (i.e., dependencies that are local to a set and dependencies that require consistency between sets). Such constraints cannot be captured by existing notions of functional, multi-valued, or join dependencies.

This paper presents the definition of NFDs and gives their meaning by translation to logic. It then presents a sound and complete set of eight inference rules for NFDs, and discusses approaches to handling the existence of empty sets in instances. Empty sets add complexity in reasoning since formulas such as $\forall x \in R.P(x)$ are trivially true when R is empty. This axiomatization represents a first step in reasoning about constraints on data warehouse applications, where both the source and target databases support complex types.

1 Introduction

Dependencies add semantics to a database schema and are useful for studying various problems such as database design, query optimization and how dependencies are carried into a view. In the context of the relational model, a wide variety of dependencies have been studied, such as functional, multivalued, join and inclusion dependencies (see [14, 2] for excellent overviews of this work). However, apart from notions of key constraints and inclusion dependencies [5, 16], dependencies in richer models than the relational model have not been as thoroughly studied.

Complex data models are, however, heavily used within biomedical and other scientific database applications. Reasoning about dependencies within these applications is becoming increasingly important as schemas get larger, queries span multiple complex databases, and new databases are created as materialized views. For example, if a new database is created as a materialized view over multiple complex databases, knowing how dependencies are carried into this complex view could eliminate expensive checking as the new database is created and later updated.

We therefore start attacking this problem by defining a notion of functional dependency for the nested relational model together with inference rules for these dependencies. We are considering the nested relational model, where set and tuple constructors are required to alternate, mainly for simplicity, but relaxing this assumption does not significantly change the inference rules. Since in this model attributes of a relation may be sets rather than atomic types, dependencies may traverse into various levels of nesting through paths. We call this new form of functional dependencies *nested functional dependencies* (NFDs).

As an example of what we would like to be able to express, consider a type *Course* defined as a set of records with attributes *cnum*, *time*, *students*, and *books*, where *students* is a set of records with labels *sid*, *age*, and *grade*, and *books* is a set of records with labels *isbn*, and *title*:

$$\begin{aligned} Course : \{<& cnum, time \\ & students : \{<sid, age, grade>\}, \\ & books : \{<isbn, title>\}>\}. \end{aligned}$$

Some nested functional dependencies that we would like to be able to express for *Course* are:

*This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO DAAH04-93-G0129, and ARPA N00014-94-1-1086.

†Partially supported by CNPq-Brazil, and Universidade Federal do Paraná.

1. *cnum* is a key.

2. Every *Course* instance is consistent on their assignment of *title* to a given *isbn*.

3. In a given course, each student gets a single grade.

4. Every *Course* instance is consistent on their assignment of *age* to *sid*.

5. A student cannot be enrolled in courses that overlap on time.

Note that there are "local" dependencies, such as dependency 3 where a student can have only one *grade* for a given course but may have different *grade*s for distinct courses. There are also "global" dependencies such as dependencies 2 and 4, where the assignment of *title* to an *isbn* and *age* to *sid* must be consistent throughout the *Course* relation. Dependency 5 illustrates how an attribute from an outer level of nesting may be determined by attributes in a deeper level of nesting. Note that even if every level of nesting presents a "key" as suggested in [1], this type of dependency is not captured by the structure of the data.

Our definition of NFDs can also be used to express other interesting properties of sets. For example, they can be used to state that some fields in a set valued attribute are required to be disjoint, or that a set is expected to be a singleton. In AceDB [18], a database which is very popular among biologists, every attribute is defined as a set. This is useful in applications where the database is sparsely populated and evolves over time, since empty sets can model optional or undefined attributes. However, some attributes can be specified to be (maximally) singleton sets. In order to reason about constraints in this model, it is therefore important to be able to express the fact that a set must be a singleton. The importance of singleton sets is also evident in [7], which investigates when functional dependencies are maintained or destroyed when relations are nested and unnested. In most cases, this relies on knowing whether a set is a singleton or multivalued.

One of the most interesting questions involving dependencies is that of logical implication, i.e., deciding if a new dependency holds given a set of existing dependencies. For functional dependencies in the relational model, this problem has been addressed from two different perspectives: a decision procedure called the tableau chase, and a sound and complete set of inference rules called Armstrong's axioms.

As an example of an inference we might want to make over the complex type *Course*, suppose we have a database *DBCourse* which is known to satisfy all the dependencies listed above. We wish to know if in *DBCourse*, given a student ID *sid*, and a *time*, there is a unique set of *books* used by the student at that time. Reasoning intuitively, the answer is affirmative since a student can be enrolled in only one course *cnum* in a given *time*, and *cnum*, which is a key, determines a set *books*. However, it would be useful to have a technique and inference rules to prove this.

The development of inference rules is important for many reasons [4]: First, it helps us gain insight into the dependencies. Second, it may help in discovering efficient decision procedures for the implication problem. Third, it provides tools to operate on dependencies. For example, in the relational model, it provides the basis for testing equivalence preserving transformations, such as lossless-join decomposition, and dependency preserving decomposition, which lead to the definition of normal forms of relations, a somewhat more mechanical way to produce a database design [19].

We therefore focus in this paper on the development of a sound and complete set of inference rules for NFDs. However, the presence of empty sets in instances causes serious problems in developing such rules since formulas such as $\forall x \in R.P(x)$ are trivially true when R is empty. We therefore initially restrict the inference problem to the case where empty sets cannot occur in any instance, and then suggest how this assumption can be relaxed by specifying where empty sets are known not to occur.

The remainder of the paper is organized as follows: Section 2 describes our nested relational model, the definition of nested functional dependencies in this model, and their translation into logic. We also contrast our approach to others taken in the literature. Section 3 presents the axiomatization of NFDs, illustrates their use on some examples, and discusses how empty sets in instances can cause problems. Section 4 concludes the paper and discusses some future work.

2 Functional Dependencies for the Nested Relation Model

The nested relational model has been well studied (see [2] for an overview). It extends the relational model by allowing the type of an attribute to be a set of records or a base type, rather than requiring it to be a base type (First Normal Form). For simplicity, we use the strict definition of the nested model and require that set and tuple constructors alternate, i.e. there are no sets of sets or tuples with a tuple component, although allowing nested records or sets does not substantially change the results established. For ease of presentation, we also assume that there are no repeated labels in a type, i.e., $<A : int, B : \{<A : int>\}>$ is not allowed.

An example of a nested relation was given by *Course* in the previous section.

More formally, a nested relational database $\mathcal{R}$ is a finite set of relation names, ranged over by $R_1, R_2, \ldots$. $\mathcal{A}$ is a fixed countable set of labels, ranged over by

$A_1, A_2, \ldots$, and $\mathcal{B}$ is a fixed finite set of base types, ranged over by $\underline{\mathsf{b}}, \ldots$

The data types **Types** are as follows:

$$\tau ::= \underline{\mathsf{b}} \mid \{\tau\} \mid \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$$

Here, $\underline{\mathsf{b}}$ are base types, e.g. boolean, integer and string. The notation $\{\omega\}$ represents a set with elements of type ω, where ω must be a record type. $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$ represents a record type with fields $A_1, \ldots, A_n$ of types $\tau_1, \ldots, \tau_n$, respectively. Each τ_i must either be a base or a set type.

A **database schema** is a pair $(\mathcal{R}, \mathcal{S})$, where $\mathcal{R}$ is a finite set of relation names, and $\mathcal{S}$ is a schema mapping $\mathcal{S} : \mathcal{R} \to$ **Types**, such that for any $R \in \mathcal{R}$, $R \overset{\mathcal{S}}{\mapsto} \tau^R$ where τ^R is a set of records in its outermost level.

A **database instance** of a database schema $(\mathcal{R}, \mathcal{S})$ is a record I with labels in $\mathcal{R}$ such that $\pi_R I$ is in $[\![\mathcal{S}(R)]\!]$ (we assume the natural denotation for types) for each $R \in \mathcal{R}$.

As an example, if $(\{Course\}, \mathcal{S})$ is a schema where

$$\begin{aligned}\mathcal{S}(Course) = \{\langle &cnum : string,\\ &time : int,\\ &students : \{\langle sid : int,\\ &\qquad grade : string\rangle\}\rangle\}.\end{aligned}$$

Then the following is an example of an instance of this schema:

$$\begin{aligned}\langle Course \mapsto \{\langle &cnum \mapsto \text{"}cis550\text{"},\\ &time \mapsto 10,\\ &students \mapsto \{\langle sid \mapsto 1001,\\ &\qquad grade \mapsto \text{"}A\text{"}\rangle,\\ &\qquad \langle sid \mapsto 2002,\\ &\qquad grade \mapsto \text{"}B\text{"}\rangle\}\rangle,\\ \langle &cnum \mapsto \text{"}cis500\text{"},\\ &time \mapsto 12,\\ &students \mapsto \{\langle sid \mapsto 1001,\\ &\qquad grade \mapsto \text{"}A\text{"}\rangle\}\rangle\}\rangle\end{aligned}$$

2.1 Nested Functional Dependencies

The natural extension of a functional dependency $X \longrightarrow A$ for the nested relational model is to allow path expressions in X and A instead of attributes. That is, X is a set of paths and A is a single path. As an example, the requirement that a student's *age* in *Course* be consistent throughout the database could be written as $Course : [students : sid \to students : age]$, where ":" indicates traversal inside a set. Note that we have enclosed the dependency in square brackets "[]" and appended the name of the nested relation, *Course*.

We start by giving a very general definition of path expressions, and narrow them to be well-defined by a given type.

Definition 2.1 *Let $\mathcal{A} = A_1, A_2, \ldots$ be a set of labels. A* **path expression** *is a string over the alphabet $\mathcal{A} \bigcup \{ : \}$. ϵ denotes the empty path. A path expression p is* **well-typed** *with respect to type τ if*

- *$p = \epsilon$, or*
- *$p = Ap'$ and τ is a record type $\langle A : \tau', \ldots \rangle$ and p' is well-typed with respect to τ', or*
- *$p = \, : p'$ and τ is a set type $\{\tau'\}$ and p' is well-typed with respect to τ'.*

As an example, $A : B$ is well-typed with respect to $\langle A : \{\langle B : int, C : int\rangle\}\rangle$, but not with respect to $\langle A : int\rangle$.

The semantics of path expressions is given by:

$$\begin{aligned}[\![\epsilon\ e]\!] &\equiv [\![e]\!]\\ [\![A\ e]\!] &\equiv [\![e]\!](A)\end{aligned}$$

$$[\![:\ e]\!] \equiv \begin{cases} \text{undefined}, & \text{if } [\![e]\!] = \{\}\\ [\![e_1]\!], & \text{otherwise, where } [\![e_1]\!] \\ & \text{is an element of } [\![e]\!]\end{cases}$$

Note that the value of a path expression that traverses into an empty set is undefined, i.e., it does not yield a value in the database domain. We say that a path expression p is **well defined** on v if it always yields a value in the database domain.

As an example, if

$$\begin{aligned}v = \langle A \mapsto \{\ &\langle B \mapsto 10, C \mapsto 20\rangle,\\ &\langle B \mapsto 15, C \mapsto 21\rangle\}\rangle\end{aligned}$$

then

- $A(v) = \{\langle B \mapsto 10, C \mapsto 20\rangle, \langle B \mapsto 15, C \mapsto 21\rangle\}$
- $A : B(v) = 10$ or $A : B(v) = 15$

To help define nested functional dependencies, we introduce the notion of path prefix.

Definition 2.2 *Path expression p_1 is a* **prefix** *of p_2 if $p_2 = p_1 p_2'$. Path p_1 is a* **proper prefix** *of p_2 if p_1 is a prefix of p_2 and $p_1 \neq p_2$.*

With this notion, we are now in a position to define nested functional dependencies (NFDs), and how an instance is said to satisfy an NFD.

Definition 2.3 *Let $\mathcal{SC} = (\mathcal{R}, \mathcal{S})$ be a schema. A* **nested functional dependency (NFD)** *over $\mathcal{SC}$ is an expression of the form $x_0 : [x_1, \ldots, x_{m-1} \to x_m]$, $m \geq 1$, such that all x_i, $0 \leq i \leq m$, are path expressions of the form $A_1^i : \ldots : A_{k_i}^i$, $k_i \geq 1$, where $x_0 = Ry$, $R \in \mathcal{R}$, and $y : x_i, 1 \leq i \leq m$, are well-typed path expressions with respect to τ^R.*

In general, the *base path* x_0 can be an arbitrary path rather than just a relation name. For the degenerate case where $m = 1$, i.e. the NFD is of form $x_0 : [\emptyset \rightarrow x_m]$, then in any value of $x_0, : x_m$ must be a constant.

Definition 2.4 *Let $f = x_0 : [x_1, \ldots, x_{m-1} \rightarrow x_m]$ be an NFD over schema SC, I an instance of SC, and v_1, v_2 two values of $x_0 : (I)$ in the database domain. I* **satisfies** *f, denoted $I \models f$, if for all v_1, v_2, whenever*

1. *$x_i(v_1) = x_i(v_2)$ for all $1 \leq i < m$, and*
2. *for every path x which is a common prefix of x_i, x_j, $1 \leq i, j \leq m$, $x(v_1)$ coincide in $x_i(v_1)$ and $x_j(v_1)$ and $x(v_2)$ coincide in $x_i(v_2)$ and $x_j(v_2)$ (i.e. x_i and x_j follow the same path up to x in v_1 and in v_2)*

then

$$x_m(v_1) = x_m(v_2)$$

If for some x_i, $1 \leq i \leq m$, $x_i(v_1)$, or $x_i(v_2)$ is undefined, we say f is trivially true.

Our definition of NFDs is very broad, and captures many natural constraints. As an example, we can precisely state the constraints on *Course* described in the introduction.

Example 2.1 In *Course*, *cnum* is a key.
Course : [*cnum* → *time*]
Course : [*cnum* → *students*]
Course : [*cnum* → *books*]

Example 2.2 For any two instances in *Course*, if they agree on *isbn* for some element of *books* then they must also agree on *title* for that element of *books*.
Course : [*books* : *isbn* → *books* : *title*]

Example 2.3 In a given course, each student gets a single grade.
Course : *students* : [*sid* → *grade*]

Note that in this example, *sid* is a "local" key to *grade*; this illustrates the use of a path rather than just a relation name outside the "[]". Contrast this to the previous example, where the NFD requires that *isbn* and *title* be consistent throughout the database.

Example 2.4 Every *Course* instance is consistent on their assignment of *age* to *sid*.
Course : [*students* : *sid* → *students* : *age*]

Example 2.5 A student cannot be enrolled in courses that overlap on time.
Course : [*time*, *students* : *sid* → *cnum*]

Some interesting properties of sets can also be expressed by NFDs. For example, if an instance I satisfies an NFD of the form $x_0 : [x_1 : x_2 \rightarrow x_1]$, then given two values v_1, v_2 of $x_0 : x_1(I)$, either $v_1 = v_2$, or $v_1 \bigcap v_2 = \emptyset$[1].

As an example, suppose that a university's courses database is defined as *Courses* : {<*school*, *scourses* : {<*cnum*, *time*>}>}, and it satisfies the NFD *Courses* : [*scourses* : *cnum* → *school*]. We can conclude that *schools* in the university do not share course numbers, because the existence of the same *cnum* in different *schools* would violate the NFD.

[1]Note that values of $x_0 : x_1(I)$ must be of set type.

A	B		E	
	C	D	F	G
1	1	3	5	6
			5	7
	C	D	F	G
2	2	2	3	4
	1	3	4	4

Figure 1: An instance that violates $R : [B : C \rightarrow E : F]$.

NFDs can also express that if a set is not empty then it must be a singleton. I.e., if an instance I satisfies an NFD of the form $x_0 : [x_1, \ldots, x_m \rightarrow x_n : A]$, where x_n is not a proper prefix of any x_i, $1 \leq i \leq m$, then for any value v of $x_0 : (I)$ in which paths $x_1 \ldots x_m$ are well-defined, all elements e of $x_n(v)$ have the same value for $A(e)$.

For example, let R be a relation with schema {<A : {<B : *int*, C : *int*>}, D : *int*>}. If $R : [D \rightarrow A : B]$, and $R : [D \rightarrow A : C]$, then it must be the case that A is either empty, or a singleton set, since for every value of A all elements agree on the values of B and C. Since these are the only attributes in A, then A has a single element.

It should be noted that our definition also allows some *unintuitive* NFDs. For example, assume R : {<A, B : {<C, D>}, E : {<F, G>}>}. Then the NFD $R : [B : C \rightarrow E : F]$ implies that:

- all tuples <F, G> in E have the same value for F when B is not empty, and
- if any tuple <C, D> in B agrees on the value of C, then the elements <F, G> in E must have the same value for F.

Figure 1 shows an instance of R that does not satisfy $R : [B : C \rightarrow E : F]$. If we only consider the first line in the table, the NFD is satisfied since all values of attribute F coincide, i.e. $B : C = 1$ determines $E : F = 5$. The existence of more than one value for F automatically invalidates the constraint because a single value in C would be related to distinct values in F as in the second line. The second line also violates the dependency because it has a value in $B : C$ that also appears in the first line, but has a different value for $E : F$.

2.2 NFDs expressed in logic

In the relational model, a functional dependency $Course : [cnum \rightarrow time, students]$ can be understood as the following formula:

$$\begin{array}{l} \forall c_1 \in Course\ \forall c_2 \in Course \\ (c_1.cnum = c_2.cnum) \rightarrow \\ (c_1.time = c_2.time\ \wedge c_1.students = c_2.students) \end{array}$$

There is also a precise translation of NFDs to logic. Intuitively, given an NFD $R : [x_1 \dots x_{m-1} \rightarrow x_m]$, we introduce two universally quantified variables for R and for each set-valued attribute in $x_1 \dots x_m$ The body of the formula is an implication where the antecedent is the conjunction of equalities of the last attributes in $x_1 \dots x_{m-1}$ and the consequence is an equality of the last attribute in x_m.

As an example, $Course : [students : sid \rightarrow students : age]$ can be translated to the following formula:

$$\begin{array}{l} \forall c_1 \in Course\ \forall c_2 \in Course \\ \forall s_1 \in c_1.students\ \forall s_2 \in c_2.students. \\ \quad (s_1.sid = s_2.sid \rightarrow s_1.age = s_2.age) \end{array}$$

To formalize this translation, we define functions *var*, and *parent*. Let $SC = (\mathcal{R}, \mathcal{S})$ be a schema, I an instance of SC, and $f = x_0 : [x_1 \dots x_{m-1} \rightarrow x_m]$ be an NFD defined over SC, where $x_i = A^i_1 : \dots : A^i_{k_i}$, $0 \le i \le m$, and $A^0_1 = R$, $R \in \mathcal{R}$.

Define *var* as a function that maps labels to variable names as follows:

- for each label A in τ^R that appears in some path x_i, $0 \le i \le m$, $var(A) = v_A$. Recall that we assume labels cannot be repeated.

The function *parent* maps a label to the variable defined for its parent as follows:

- for all A^i_1, $1 \le i \le m$, $parent(A^i_1) = var(A^0_{k_0})$, i.e., the parent of the first labels in paths $x_1 \dots x_m$ is the variable associated with the last label in path x_0.
- $parent(A^i_{j+1}) = var(A^i_j)$. Let $\{A^*_1 \dots A^*_q\}$ be the set of such A_j labels, i.e., the set of labels that have some descendent in a path expression.

Also, let $parent(A^0_1).A^0_1 = R$. Then f is equivalent to the following logic formula:

$$\begin{array}{l} \forall v_{A^O_1} \in parent(A^0_1).A^0_1\ \dots \\ \forall v_{A^0_{k_0-1}} \in parent(A^0_{k_0-1}).A^0_{k_0-1} \\ \quad \forall v^1_{A^0_{k_0}} \in parent(A^0_{k_0}).A^0_{k_0}\ \ \forall v^2_{A^0_{k_0}} \in parent(A^0_{k_0}).A^0_{k_0} \\ \quad\quad \forall v^1_{A^*_1} \in parent(A^*_1)^1.A^*_1\ \forall v^2_{A^*_1} \in parent(A^*_1)^2.A^*_1 \dots \\ \quad\quad \forall v^1_{A^*_q} \in parent(A^*_q)^1.A^*_q\ \forall v^2_{A^*_q} \in parent(A^*_q)^2.A^*_q \\ \quad\quad\quad ((true\ \wedge \\ \quad\quad\quad parent(A^1_{k_1})^1.A^1_{k_1} = parent(A^1_{k_1})^2.A^1_{k_1} \wedge \dots \wedge \\ \quad\quad\quad parent(A^{m-1}_{k_{m-1}})^1.A^{m-1}_{k_{m-1}} = parent(A^{m-1}_{k_{m-1}})^2.A^{m-1}_{k_{m-1}}) \\ \quad\quad\quad \rightarrow \\ \quad\quad\quad (parent(A^m_{k_m})^1.A^m_{k_m} = parent(A^m_{k_m})^2.A^m_{k_m})) \end{array}$$

Note that only one variable is mapped to each label in $A^0_1, \dots, A^0_{k_0-1}$, whereas two variables are used elsewhere.

Using this translation, examples 2. 2 and 2. 3 can be expressed as:

- $Course : [books : isbn \rightarrow books : title]$
  $\forall c_1 \in Course\ \forall c_2 \in Course$
  $\forall b_1 \in c_1.books\ \forall b_2 \in c_2.books.$
  $\quad (b_1.isbn = b_2.isbn \rightarrow b_1.title = b_2.title)$

  Note that *books* is referred to twice in the dependency, but that only two variables for *books* are introduced in the logical form.

- $Course : students : [sid \rightarrow grade]$
  $\forall c \in Course$
  $\forall s_1 \in c.students\ \forall s_2 \in c.students$
  $\quad (s_1.sid = s_2.sid \rightarrow s_1.grade = s_2.grade)$

  Note that only one variable is introduced for labels in x_0 (except for the last label), and that two variables are introduced for all other labels.

2.3 Discussion

In the definition of NFDs, the base path can be an arbitrary path rather than just a relation name. The motivation for allowing this is to syntactically differentiate between local and global functional dependencies: $R : A : [B \rightarrow C]$ is a *local* functional dependency in A, while $R : [A : B \rightarrow A : C]$ defines a **global** dependency between B and C. However, the local dependency is provably equivalent[2] to the dependency $R : [A,\ A : B \rightarrow A : C]$. Intuitively, by requiring equality on A (as a set), the dependency between B and C becomes local to the set. Therefore, the expressive power of NFDs with arbitrary paths and relation names as base paths are the same. However, we believe that the first form is more intuitive.

Most of the early work on functional dependencies for the nested relational model either used the definition of functional dependencies given for the relational model [15], or proposed a simple extension to allow equality on sets [12]. Our definition clearly subsumes these definitions.

The idea of extending functional dependencies to allow path expressions instead of simple attribute names has been investigated by Weddell [21] in the context of an object-oriented data model. While this work supports a data model of classes, where each class is associated with a simple type (a flat record type), our model supports a nested relational model

[2]The equivalence of these two forms is proved in the next section.

with arbitrary levels of nesting. In [21], following a path entails an implicit "dereference" operation, while in NFDs following a path means traversal into an element of a nested set. We believe these two works are complementary and that it would be interesting to investigate how the two approaches could be combined into a single framework.

3 Inference Rules for NFDs Without Empty Sets

One of the most interesting questions involving NFDs is that of logical implication, i.e., deciding if a new dependency holds given a set of existing dependencies. This problem can be addressed from two perspectives: One is to develop algorithms to decide logical implication, for example, tableau chase techniques (see [13] for the relational model, and more recently [16, 17] for a complex object model). The other is to develop inference rules that allow us to derive new dependencies from the given ones.

In this section, we present a sound and complete set of inference rules for NFDs in the restricted case in which no empty sets are present in any instance. The extension to allow empty sets in instances is discussed in detail in Section 3.2.

The implication problem for NFDs that we are considering is therefore defined as:

Definition 3.1 *Let SC be a schema, Σ be a set of NFDs over SC, and σ an NFD over SC. Σ logically implies σ under SC, denoted $\Sigma \models_{SC} \sigma$ if for all instances I of SC* **with no empty sets**, *$I \models \Sigma$ implies $I \models \sigma$.*

3.1 NFD Rules

Conceptually, the NFD rules can be broken up into three categories: The first three mirror Armstrong's axioms – reflexivity, augmentation and transitivity. The next two – push-in and pull-out – transform between the alternate forms of NFDs discussed at the end of the last section.[3] The last three rules allow inferences based solely on the nested form of the data – locality, singleton, and prefix.

In the following, $x, y, z, x_0, x_1, \ldots$ are path expressions, and $A_1, A_2, \ldots, B_1, B_2, \ldots$ are attribute labels. XY denotes $X \bigcup Y$, where X, Y are sets of path expressions, and $x : X$ denotes the set $\{x : x_1, \ldots x : x_k\}$, where $X = \{x_1, \ldots, x_k\}$.

The **NFD-rules** are:

- **reflexivity**:
  if $x \in X$ then $x_0 : [X \to x]$.
- **augmentation**:
  if $x_0 : [X \to z]$ then $x_0 : [XY \to z]$.
- **transitivity**:
  if $x_0 : [X \to x_1], \ldots, x_0 : [X \to x_n]$,
  $x_0 : [x_1, \ldots, x_n \to y]$
  then $x_0 : [X \to y]$.
- **push-in:**
  if $x_0 : y : [X \to z]$ then $x_0 : [y,\ y : X \to y : z]$
- **pull-out:**
  if $x_0 : [y,\ y : X \to y : z]$ then $x_0 : y : [X \to z]$
- **locality:**
  if $x_0 : [A : X,\ B_1, \ldots, B_k \to A : z]$
  then $x_0 : A : [X \to z]$.
- **singleton**: if
  1. $x_0 : [x \to x : A_1], \ldots, x_0 : [x \to x : A_n]$
  2. type of x is $\{<A_1, \ldots A_n>\}$

  then $x_0 : [x : A_1, \ldots, x : A_n \to x]$
- **prefix:** if
  1. $x_0 : [x_1 : A,\ x_2, \ldots, x_k \to y]$
  2. x_1 has one or more labels
  3. x_1 is not prefix of y

  then $x_0 : [x_1,\ x_2, \ldots, x_k \to y]$

Theorem 3.1 *Let SC be a schema. The NFD-rules are sound and complete for logical implication of NFDs under SC for the case when no empty sets are present in any instance.*

Proof Outline: Soundness can be easily verified. For completeness, we assume that the domain of all base types are infinite, and use the standard form of proof. Let $SC = (\mathcal{R}, \mathcal{S})$. We introduce the notion of a closure of a set of path expressions X with respect to a base path x_0 and a set of NFDs Σ, denoted as $(x_0, X, \Sigma)^*$, as the set of paths $x_0 : q$ such that $x_0 : q$ is a well-typed path with respect to some $R \in \mathcal{R}$ and $x_0 : [X \to q]$ can be derived from the NFD-rules. Then an instance I is built such that I satisfies Σ, but not $x_0 : [X \to y]$ if $x_0 : y \notin (x_0, X, \Sigma)^*$. The construction of I is described in Appendix A.

As an example of the use of the NFD-rules, let R be a relation with schema $\{<A : \{<B : \{<C>\}, E : \{<F, G>\}>\}, D>\}$, on which the following NFDs are defined:
(nfd1) $R : [A : B : C,\ D \to A : E : F]$
(nfd2) $R : A : [B \to E : G]$

We claim that $R : A : [B \to E]$. The proof is as follows:

[3] A discussion of why we don't adopt a simpler form of NFDs which would eliminate these two rules is deferred to Section 3.2.

1. $R : A : [B : C \rightarrow E : F]$ by locality of nfd1.

   The locality rule allows us to derive a local NFD from a global one by dismissing the attributes outside the level of nesting of the local NFD. In the example above, note that for any element in R, given a value of A there exists a unique value of D, since they are labels in a record type. Therefore, locally for any value of A, $B : C \rightarrow E : F$.

2. $R : A : [B \rightarrow E : F]$ by prefix rule on (1).

   (1) states that whenever two tuples in R have a common value for C in the set B, then the value of $E : F$ must also agree. In particular, if two tuples agree on the value of B then they present a common element, since we assumed that there are no empty sets in instances of R.

3. $R : A : E : [\emptyset \rightarrow F]$ by locality of (2).

   If in any tuple in $R : A$ the value of B determines the value of $E : F$, then all elements in E must agree on the value of F, otherwise (2) would be violated. Therefore, locally in any $A : E$ the value of F is constant.

4. $R : A : [E \rightarrow E : F]$ by push-in.

   If the value of F is constant inside any value of $A : E$, then for any given value of $A : E$ there exists a unique value of F. Therefore, the whole set determines the value of F.

5. $R : A : E : [\emptyset \rightarrow G]$ by locality of nfd2.

6. $R : A : [E \rightarrow E : G]$ by push-in.

7. $R : A : [E : F,\ E : G \rightarrow E]$ by singleton with (4) and (6).

   Since the value of the set E determines the value of each of its attributes, then E must be a singleton. Therefore, the values of its unique element determines the value of the set.

8. $R : A : [B \rightarrow E]$ by transitivity with (7), (2), and nfd2.

3.2 Discussion

Simple NFDs. Note that push-in and pull-out simply change between equivalent forms of NFDs. I.e., an NFD of form $R : y : [x_1, \ldots, x_k \rightarrow z]$ is equivalent to $R : [y,\ y : x_1, \ldots, y : x_k \rightarrow y : z]$. Therefore, we could change the definition of an NFD to allow only relation names as the base path (x_0) of an NFD, without changing its expressive power.

In this simpler form of NFDs, it can be shown that there are only six inference rules: push-in and pull-out are unnecessary. Of the remaining rules, only locality must be modified to what we call **full-locality**: if

1. $x_0 : [x : X,\ Y \rightarrow x : z]$

2. x is not a proper prefix of any $y \in Y$

then $x_0 : [x,\ x : X \rightarrow x : z]$.

Note that full-locality combines the pull-out and locality rules. As an example of the need to use full-locality rather than locality, consider the following:

Example 3.1 Let f_1 be the NFD $R : [A : B : C,\ A : D \rightarrow A : B : E]$. Applying the locality rule, we can get $R : [A,\ A : B : C,\ A : D \rightarrow A : B : E]$, but not $R : [A : B,\ A : B : C \rightarrow A : B : E]$. The latter is derivable using full-locality.

Although the simpler form of NFDs yields a smaller set of axioms, we believe that the first form, which allows an arbitrary base path, is more intuitive since it makes a syntactic distinction between inter- and intra-set dependencies.

The Problem of Empty Sets. As mentioned earlier, the presence of empty sets causes difficulties in reasoning since formulas such as $\forall x \in R.P(x)$ are trivially true when R is empty. In particular, the transitivity rule is no longer sound in the presence of empty sets, as illustrated below.

Example 3.2 The instance of R below satisfies $R : [A \rightarrow B : C]$, $R : [B : C \rightarrow D]$, but not $R : [A \rightarrow D]$.

A	B	D	E
1	$\emptyset$	2	3
1	$\emptyset$	3	4
2	{<C : 3>}	4	5

One reasonable solution to this problem is to disallow empty sets only in certain portions of the schema; this is analogous to specifying NON-NULL for certain attributes in a relational schema. The transitivity rule can then be modified to reason about where empty sets are known not to occur. We do this by introducing a new relation *follow* between paths.

Definition 3.2 *Path expression* p_1 **follows** p_2 *if* $p_1 = p'_1 A$*, and* p'_1 *is a proper prefix of* p_2.

Intuitively, p_1 follows p_2 if it only traverses the set-valued attributes traversed by p_2. For example, a path A follows any path p, $|p| \geq 1$, since $A \equiv \epsilon A$, and ϵ is a proper prefix of any path. A path $A : B$ follows $A : B$, $A : C : D$, but not A, E, and $F : G$.

The new transitivity rule is then defined as: if

1. $x_0 : [X \rightarrow x_1], \ldots, x_0 : [X \rightarrow x_n]$,
   $x_0 : [x_1, \ldots, x_n \rightarrow y]$

2. for all p in $\{x_1, \ldots, x_n\} - X$, if p does not follow y, then p is known not to be an empty set

then $x_0 : [X \rightarrow y]$.

The fact that transitivity does not generally hold in the presence of empty sets has also influenced our definition of NFDs to allow only single paths on the right-hand side of functional dependencies rather than sets of paths.

Recall that in the relational model, a functional dependency (FD) $X \rightarrow Y$, where X, Y are sets of attributes, can be decomposed into a set of FDs with single attributes on the right-hand side of the implication. Unfortunately, the decomposition rule follows from reflexivity and transitivity and cannot therefore be uniformly applied with NFDs in the presence of empty sets.

The presence of empty sets also affects the prefix rule. Consider the instance I presented in Example 3. 2. Notice that I satisfies $R : [B : C \rightarrow E]$, but not $R : [B \rightarrow E]$. A modified prefix rule to take this into account is: if

1. $x_0 : [x_1 : A,\ x_2, \ldots, x_k \rightarrow y]$
2. x_1 has one or more labels, and x_1 is not prefix of y
3. x_1 is not an empty set

then $x_0 : [x_1,\ x_2, \ldots, x_k \rightarrow y]$

4 Conclusion

We have presented a definition of functional dependencies (NFD) for the nested relation model. NFDs naturally extend the definition of functional dependencies for the relational model by using path expressions instead of attribute names. The meaning of NFDs was given by defining their translation to logic.

NFDs provide a framework for expressing a natural class of dependencies in complex data structures. Moreover, they can be used to reason about constraints on data integration applications, where both sources and target databases support complex types.

We presented a set of inference rules for NFDs that are sound and complete for the case when no empty sets are present. Although for simplicity we have adopted the nested relational model, and the syntax of NFDs is closely related to this model, allowing nested records or sets would not change the inference rules presented significantly. However, new rules would have to be added to consider path expressions of record types as the current syntax only allows path expressions of set and base types. As an example, we would need a rule that states that if in $R\ x$ is a path of type $<A_1, \ldots, A_n>$, then $R : [x.A_1 \ldots x.A_n \rightarrow x]$, where "." indicates record projection.

In [7], Fischer, Saxton, Thomas and Van Gucht investigate how nesting defined on a normalized relation destroys or preserves functional and multivalued dependencies; they also present results on the interaction of inter- and intra-set dependencies. Their results are based on case studies of the cardinality of relations, and of the containment relation between the set of attributes over which the nesting is defined and the set of attributes involved in the dependency. Many results depend on the fact that a nested relation is a singleton set. In our definition of NFDs, both inter- and intra-set dependencies can be expressed. NFDs can also express that a given set is expected to be a singleton. As a result, our work generalizes their results by providing a general framework to reason about interactions between nesting and functional dependencies.

In future work, we intend to investigate a relaxation of the assumption that no empty sets are present in any instance, by requiring the user to define which set-valued paths are known to have at least one element. We believe this is a natural requirement to make, since definition of cardinality has long been recognized as integral part of schema design [6] and is part of the DDL syntax for SQL (NON-NULL). Generalizing the inference rules to this case would allow us to reason about constraints for a larger family of instances.

The tableaux chase has been used as a decision procedure for functional dependencies in the relational model [13], and extended to determine logical implication on views [10]. We are currently working on an extension of the tableau technique to determine view dependencies for NFDs. A definition of tableaux for the nested relational model has been proposed in [16], and we are using it to develop a transformation rule to chase nested tableaux with NFDs. We believe that the insights gained with the axiomatization presented in this paper will be of significant benefit.

Acknowledgments

The authors thank Val Tannen, Peter Buneman and Dan Suciu for numerous fruitful discussions and suggestions.

References

[1] S. Abiteboul, N. Bidoit. "Non first normal form relations: An algebra allowing restructuring". *Journal of Computer and System Sciences*, 33(3): 361-390, 1986.

[2] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

[3] A.V. Aho, Y. Sagiv, J.D. Ullman. "Equivalences among relational expressions". *SIAM Journal of Computing*, 8(2):218-246, May 1979.

[4] C. Beeri, M.V. Vardi. "Formal systems for tuple and equality generating dependencies". *SIAM Journal of Computing*, 13(1):76-98, February 1984.

[5] P. Buneman, W. Fan, S. Weinstein. "Path Constraints on Semistructured and Structured Data". In *Proceedings of the Seventeenth Symposium on Principles of Database Systems*, 1998.

[6] P.P. Chen. "The entity-relationship model - Toward a unified view of data". *ACM Transactions on Database Systems*, 1:9-36, 1976.

[7] P.C. Fischer, P.C., L.V. Saxton, S.J. Thomas, D. Van Gucht. "Interactions between Dependencies and Nested Relational Structures". *Journal of Computer and System Sciences*, 31: 343-354, 1985.

[8] C. Hara, S. Davidson. "Inference rules for nested functional dependencies". Technical Report MS-CIS-98-19. University of Pennsylvania. 1998.

[9] A. Klug. "Calculating Constraints on Relational Expressions". *ACM Transactions on Database Systems*, 5(3):260-290, September 1980.

[10] A. Klug, R. Price. "Determining View Dependencies Using Tableaux". *ACM Transactions on Database Systems*, 7(3):361-380, September 1982.

[11] A. Kosky. *Transforming Databases with Recursive Data Structures.* Ph.D. Thesis, University of Pennsylvania, 1996.

[12] A. Makinouchi. "A consideration on normal form of not-necessarily-normalized relation in the relational data model". In *Proceedings of the International Conference on Very Large Databases*, pp. 447-453, 1977.

[13] D. Maier, A. Mendelzon, Y. Sagiv. "Testing Implications of Data Dependencies". ACM Transactions on Database Systems, 4(4): 455-469, December 1979.

[14] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Inc., 1983

[15] Z.M. Ozsoyoglu, L.-Y. Yuan. "A new normal form for nested relations". ACM Transactions on Database Systems, 12(1):111-136, March 1987.

[16] L. Popa. "A Language for Nested Tableaux". draft. University of Pennsylvania, 1998.

[17] L. Popa, V. Tannen. "An Equational Chase for Path-Conjunctive Queries, Constraints, and Views". Proceedings of ICDT'99.

[18] J. Thierry-Mieg, R. Durbin. "Syntactic Definitions for the ACEDB Data Base Manager". Technical report, MRC Laboratory for Molecular Biology, Cambridge. 1992.

[19] J.D. Ullman. *Principles of Database Systems*, Second Edition. Computer Science Press, 1983.

[20] G. Weddell. "A theory of functional dependencies for object-oriented data models". In *Deductive an Object-Oriented Databases*, Eds. W. Kim, J.-M. Nicolas, S. Nishio, Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 165-184.

[21] G. Weddell. "Reasoning about Functional Dependencies Generalized for Semantic Data Models". ACM Transactions on Database Systems, 17(1): 32-64 , March 1992.

[22] L. Wong. *Querying Nested Collections.* Ph.D. Thesis, University of Pennsylvania, 1994.

[23] M. Zloof. "Query-by-Example: the invocation and definition of tables and forms". In *Proceedings of ACM International Conference on Very Large Databases*, pp. 1-24, September 1975.

A Completeness of NFDs

In this section we describe the construction of the instance I that is the basis for the completeness proof. First, we need to define the set of paths in a schema.

Definition A.1 *Let $SC = (\mathcal{R}, \mathcal{S})$ be a schema. Then the* **paths** *of SC, denoted as $Paths(SC)$, is the set of all path expressions $p \equiv Rp'$, such that $R \in \mathcal{R}$, and p' is well-typed with respect to τ^R. Similarly, the* **paths** *of R, $R \in \mathcal{R}$, denoted as $Paths_{SC}(R)$, is the set of paths p such that $p \in Paths(SC)$, and $p \equiv Rp'$.*

Let $SC = (\mathcal{R}, \mathcal{S})$ be a schema, Σ a set of NFDs over SC, X a set of paths such that $X \subseteq Paths(R)$, $R \in \mathcal{R}$, and x_0 a path in $Paths(R)$. The construction of an instance I of R such that $I \models \Sigma$, but $I \not\models x_0 : [X \rightarrow x]$ if $x_0 : x \notin (x_0, X, \Sigma)^{*,SC}$ is as follows. We assume that the domain of all base types are infinite, and to make the exposition simpler, we consider a unique base type b in our data model.

Construction of I: Let *closure* be $(x_0, X, \Sigma)^{*,SC}$, where $x_0 \equiv Rx'_0$. $value(p)$ are global variables. If p is a set of records and in its construction $value(p')$ is used (this happens when p is prefix of p') then $value(p')$ should be thought as a placeholder until its value is evaluated.

```
$val := newValue();$
for all $p \in closure$
    $value(p) := assignVal(val, p);$
$I := assignX_0(R);$
```

The auxiliary functions are defined as:

newValue(): returns a fresh new value in the domain of b.

assignX_0(p): it is a function that starts the construction of instance I by assigning new fresh values to every path that is not a prefix of x_0. r is a local variable of type $<A_1, \dots A_n>$, where type of p is $\{<A_1, \dots A_n>\}$.

```
if $p = x_0$ then return $assignVal(0, x_0);$
for each $A_i$, $1 \leq i \leq n$
    if $p : A_i$ is prefix of $x_0$ then
        $r.A_i := assignX_0(p : A_i);$
    else
        $r.A_i := assignNew(p : A_i);$
return $\{r\};$
```

assignVal (val, p): it is a function that gives a value *val* for a path p depending on the type of p in a schema SC. r_1, and r_2 are local variables of type t, where the type of p is $\{t\}$.

```
if $type_{SC}(p) = b$ then return $val$;
if $type_{SC}(p) = \{b\}$ then return $\{val\}$;
if $type_{SC}(p) = \{<A_1, \dots, A_n>\}$ then
    for all $A_i$, $1 \leq i \leq n$
        if $p : A_i \in closure$ then
            $r_1.A_i := value(p : A_i)$;
            $r_2.A_i := value(p : A_i)$;
        else
            $r_1.A_i := assignNew(p : A_i)$;
            $r_2.A_i := assignNew(p : A_i)$;
    return $\{r_1, r_2\}$;
```

assignNew (p): it is a function that gives a new fresh value for a path p, $p \notin closure$, depending on the type of p in a schema SC. If type of p is $\{t\}$, then r is a local variable of type t.

```
if $type_{SC}(p) = b$ then return $newValue()$;
if $type_{SC}(p) = \{b\}$ then return $\{newValue()\}$;
if $type_{SC}(p) = \{<A_1, \dots, A_n>\}$ then
    for all $A_i$, $1 \leq i \leq n$
        if $p : A_i \in closure$ then
            $r.A_i := value(p : A_i)$
        else $r.A_i := assignNew(p : A_i)$
    if $\{p : A_1, \dots, p : A_n\} \subseteq closure$ then
        return $\{r, newRow(p, (p, \emptyset)^*)\}$
    else
        return $\{r\}$
```

newRow(p, sameVal): The type of p is $\{<A_1, \dots A_n>\}$, where $p \notin closure$, and for all A_i, $1 \leq i \leq n$, $p : A_i \in closure$. This function returns a record, where the value of A_i, $1 \leq i \leq n$ is set to $value(p : A_i)$ if $p : A_i \in sameVal$, otherwise A_i is given a new fresh value. r is a local variable of type $<A_1, \dots A_n>$

```
for all label $A_i$, $1 \leq i \leq n$
    if $p : A_i \in sameVal$ then
        $r.A_i := value(p : A_i)$;
    else
        if $type_{SC}(p : A_i) = b$ then
            $r.A_i := newValue()$;
        if $type_{SC}(p : A_i) = \{b\}$ then
            $r.A_i := \{newValue()\}$;
        if $type_{SC}(p : A_i) = \{<B_1, \dots, B_k>\}$ then
            $r.A_i := \{newRow(p : A_i, sameVal)\}$;
return $r$;
```

Lemma A.1 *Let X be a set of paths, x_0 a path, and Σ a set of NFDs. If I is an instance built as described above, then $I \models \Sigma$, and $I \not\models x_0 : [X \rightarrow y]$ if $x_0 : y \notin (x_0, X, \Sigma)^*$.*

To illustrate the algorithm described consider the following examples.

Example A.1 Let R be a relation with schema $\{<A, B : \{<C>\}, D, E : \{<F, G>\}, H : \{<J, L>\}, I, M : \{<N, O>\}>\}$. The set Σ of NFDs defined for R are:

$$\begin{array}{l} R : [A \rightarrow B : C] \\ R : [B : C \rightarrow D] \\ R : [D \rightarrow E : F] \\ R : [A \rightarrow E : G] \\ R : [B : C \rightarrow H] \\ R : [I \rightarrow H : J] \end{array}$$

Then, $(R, \{B\}, \Sigma)^* = \{R : B, R : B : C, R : D, R : E : F, R : H, R : H : J\}$. The following instance is constructed using the algorithm presented.

A	B	D	E		H		I	M	
	C		F	G	J	L		N	O
3	0	0	0	5	0	1	$\{7\}$	9	10
					0	2			
	C		F	G	J	L		N	O
4	0	0	0	6	0	1	$\{8\}$	11	12
					0	2			

Example A.2 Let R be a relation with schema $\{<A : \{<B : \{<C, D, E : \{<F, G>\}>\}>\}, H>\}$. The set Σ of NFDs defined for R are:

$$\begin{array}{l} R : [A : B : C \rightarrow A : B] \\ R : [A : B : C \rightarrow A : B : E : F] \\ R : [H \rightarrow A : B : D] \end{array}$$

Then, $(R, \{A : B : C\}, \Sigma)^* = \{R : A : B : C, R : A : B, R : A : B : D, R : A : B : E : F\}$. The following instance is constructed using the algorithm presented.

A				H
B				
C	D	E		
		F	G	
0	0	0	1	11
		F	G	
0	0	0	2	
B				
C	D	E		
		F	G	
3	0	5	6	
B				
C	D	E		
		F	G	
0	0	0	1	12
		F	G	
0	0	0	2	
B				
C	D	E		
		F	G	
7	0	9	10	