Implication Problems for Functional Constraints on Databases Supporting Complex Objects*

Minoru ITO
Department of Information and Computer Science
Faculty of Engineering Science
Osaka University
Toyonaka, Osaka 560, Japan

Grant E. WEDDELL
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

*This research was supported in part by a grant from the Ministry of Education, Science and Culture of Japan, by the Natural Sciences and Engineering Research Council of Canada, and by Bell-Northern Research Ltd.

Abstract

Virtually all semantic or object-oriented data models assume objects have an identity separate from any of their parts, and allow users to define complex object types in which part values may be any other objects. In [20], a more general form of functional dependency is proposed for such models in which component attributes may correspond to descriptions of property paths, called *path functional dependencies* (PFDs). The main contribution of the reference is a sound and complete axiomatization for PFDs when databases may be infinite. However, a number of issues were left open which are resolved in this paper. We first prove that the same axiomatization remains complete if PFDs are permitted empty left-hand sides, but that this is *not* true if logical consequence is defined with respect to finite databases. We then prove that the implication problem for arbitrary PFDs is decidable. The proof suggests a means of characterizing an important function closure which is then used to derive an effective procedure for constructing a deterministic finite state automation representing the closure. The procedure is further refined to efficient polynomial time algorithms for the implication problem for cases in which antecedent PFDs are a form of complex key constraint.

Index Terms: constraints, functional dependencies, object-oriented data models, complex objects, implication problems

1. Introduction

There are at least two problems with the relational model when used for more involved applications [10]: users must introduce properties of objects to serve as their means of reference, and all relationships between objects must be expressed indirectly in terms of these properties. Virtually all semantic or object-oriented data models overcome these problems by assuming that objects have an identity separate from any of their parts, and by allowing users to define complex object types in which part values may be any other objects [1, 2, 11, 14, 16]. A more general language of functional constraints for a data model supporting the definition of such complex object types was considered in [20]. One feature of the model in common with a number of others [4, 5, 8] is that a database is viewed as a directed labeled graph. The idea is that objects and property values correspond to vertices and arcs respectively. The constraint language is novel since it allows descriptions of property value paths in a database graph to occur as component attributes. Members of the language are therefore referred to as *path functional dependencies* (PFDs).

An example of a collection of complex object types which can be defined in terms of the data model in [20] is illustrated by the UNIVERSITY *schema graph* in Figure 1, which characterizes information about student course enrollment at a hypothetical university. Informally, each complex object type is represented by a labeled vertex together with a number of outgoing labeled arcs. The vertex label is a *class* name and each outgoing arc represents a function which is total on the "from" class and single-valued on the "to" class.

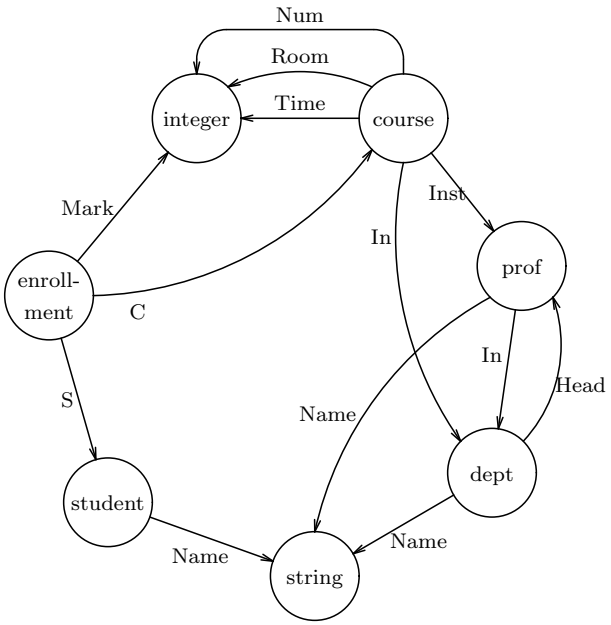

Figure 1: The UNIVERSITY schema graph.

Some examples of PFDs over the UNIVERSITY schema are listed in Table 1. The initial four entries use the special property value path descriptor, `Id`, to assert "keys." For example, the first is satisfied by a database graph only if no two departments have the same name.

(Similar constraints might also be given for students and professors.) The fifth and sixth PFDs are consequences of a requirement that professors only teach courses offered by their own departments, while the last is justified by physical reality—it asserts that a student cannot be enrolled in two separate courses at the same time. Note that the last may also be viewed as a form of complex or embedded key constraint. This becomes more apparent if one considers an alternative wording for the constraint: "in the context of the enrollments for a particular student, no two courses are given at the same time."

Table 1: PFDs over the UNIVERSITY schema.

dept(Name → Id)
dept(Head → Id)
course(Num In → Id)
enrollment(S C → Id)
course(Inst.In → In)
course(In → Inst.In)
enrollment(S C.Time → C)

There are many reasons why it is important to be able to reason about functional dependencies beyond their use in relational schema design and evaluation. An early application in query optimization involves determining *minimal covers* of selection and join conditions [3]. Several authors have also suggested how they may be used to aid in automatically inserting "cut" operators in access plans based on nested iteration [9, 12, 13, 20], in detecting search conditions for complex object indices [20], and in deducing when "project" operations (or DISTINCT modifiers) can be eliminated from a query expression [20].

An example of the last case, from [17], will help to motivate some of the results in this paper. To begin, consider the following query on the UNIVERSITY database.

> *Find all students enrolled in some course taught at the same time as some other course numbered 101 that is taught by a professor in the CS department.*

An access plan for the request, expressed in terms of a complex object algebra [7, 8, 15, 18], might be given as follows.

$$\texttt{T1} := \sigma_{\text{Inst.In.Name='CS'} \wedge \text{Num=101}}\,\texttt{course}$$
$$\texttt{T2} := \texttt{T1} \bowtie_{\text{Time=C.Time}} \texttt{enrollment}$$
$$\texttt{T3} := \pi_{\{\text{S}\}}\texttt{T2}$$

The problem is to determine if it is possible for the number of tuples in `T2` to ever exceed the number of tuples in `T3`. To see how PFD theory can help solve the problem, consider an abstraction of the query as the additional `result` object type on the UNIVERSITY schema illustrated by Figure 2. In addition, include in the query abstraction the following list of four PFDs.

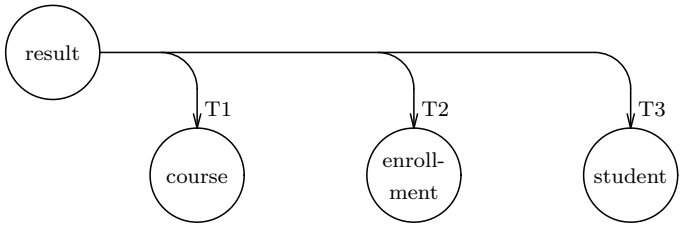

Figure 2: A QUERY ON THE UNIVERSITY SCHEMA.

```
result( T3 → T2.S )
result( T1.Time → T2.C.Time )
result( ∅ → T1.Inst.In.Name T1.Num )
result( T1 T2 T3 → Id )
```

Each is mandated in turn by the projection, join and selection operators, and on the grounds that any *particular* combination of `T1`, `T2` and `T3` tuples need only be recorded at most once by a `result` object. The issue is clearly resolved if the key PFD

```
result( T3 → Id )
```

is a logical consequence of these and the other PFDs listed in Table 1. In fact, the set of inference axioms proposed in [20] is sufficient to determine that this is indeed the case.

The main contribution of this earlier work is a proof that the inference axioms are complete. However, the proof of completeness depended on two assumptions: that the left-hand sides of antecedent PFDs are non-empty, and that databases can be of infinite size. In Section 3, we prove a positive and a negative result concerning these assumptions. The positive result is that allowing PFDs with empty left-hand sides does not alter the theory. (The example above demonstrates at least one use of such PFDs in abstracting selection conditions in queries.) The negative result is that the inference axioms are *not* complete if logical consequence is defined with respect to finite databases only; that is, we prove that the implication problem and finite implication problem for PFDs are not equivalent.

Our main result relates to another issue which was left open in [20]. In Section 4, we prove that the implication problem for arbitrary PFDs is decidable, which we believe to be important new evidence that PFDs are a feasible concept in complex object databases. The proof suggests a means of characterizing an important function closure. In Section 5, we derive an effective procedure for constructing a deterministic finite state automation representing the closure. The procedure is further refined in Section 6, in which we derive polynomial time algorithms for the implication problem for cases in which antecedent PFDs are key constraints. Our summary comments are given in Section 7.

2. Definitions and basic concepts

We begin by presenting the syntax of our data model, commonly referred to as the *data definition language* (DDL). An instance of the DDL defines a space of possible databases. In

our case, an element of this space will correspond to a labeled directed graph.

Definition 1: (*syntax—the* DDL) A *class schema* S consists of a finite set of complex object types of the form

$$C\{P_1 : C_1, \ldots, P_n : C_n\}$$

in which C is a class name, and the set $\{P_1, \ldots, P_n\}$ are its properties, written $\mathit{Props}(C)$. Each property P_i is unique in a given class scheme, and its type, written $\mathit{Type}(C, P_i)$, is the name C_i of another (not necessarily distinct) class scheme. The set of names of classes in S is denoted $\mathit{Classes}(S)$. By convention, only the first letter of property names will be capitalized. □

The declarations for a UNIVERSITY class schema outlined pictorially in Figure 1 are formally defined in Table 2. Note how several properties, such as `S` or `C`, have non-built-in classes as their range, and how the `In` and `Head` properties demonstrate that problem schema may be cyclic.

Table 2: The UNIVERSITY schema.

```
enrollment{ S: student, C: course, Mark: int }
student{ Name: string }
course{ In: dept, Inst: prof, Room: int, Num: int, Time: int }
dept{ Name: string, Head: prof }
prof{ In: dept, Name: string }
string{ }
int{ }
```

Definition 2: (*semantics—a database*) A database for class schema S is a (possibly infinite) directed graph $G(V, A)$ with vertex and edge labels corresponding to class and property names respectively. G must also satisfy the following three constraints, where the class name label of a vertex v is denoted $l_{Cl}(v)$.

1. (*property value integrity*) If $u \xrightarrow{P} v \in A$, then $P \in \mathit{Props}(l_{Cl}(u))$ and $l_{Cl}(v) = \mathit{Type}(l_{Cl}(u), P)$.
2. (*property functionality*) If $u \xrightarrow{P} v, u \xrightarrow{P} w \in A$, then $v = w$.
3. (*property value completeness*) If $u \in V$, then there is an arc $u \xrightarrow{P} v \in A$ for every $P \in \mathit{Props}(l_{Cl}(u))$. □

The UNIVERSITY schema graph in Figure 1 is one possible database for the UNIVERSITY schema. In this case, a single object exists for each complex object type. The directed graph of Figure 3 depicts another possibility in which two departments have the same name. (Note that different `string` vertices represent different strings, although the particular strings involved, or integers for that matter, are never important to our presentation.)

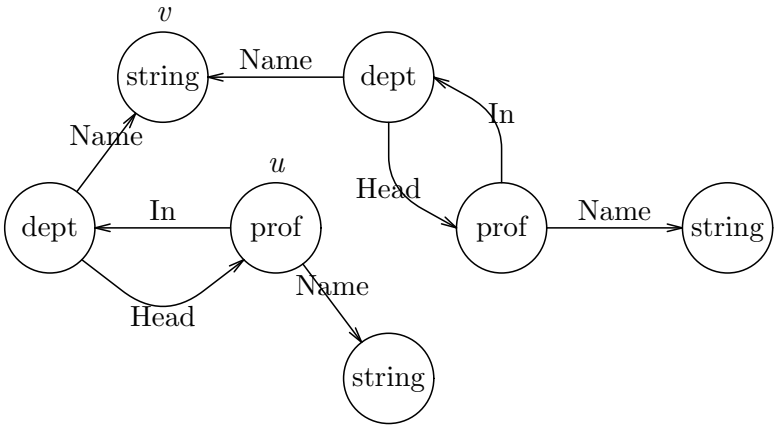

Figure 3: A DATABASE FOR THE UNIVERSITY SCHEMA.

Definition 3: A *path function pf* over schema S is either `Id` (short for *identity*), or a finite sequence of property names separated by dots. (We assume `Id` does not correspond to the name of any property in S. The identity path function is our means of referring to property value paths of zero length.) Their composition and length are defined as follows.

$$pf_1 \circ pf_2 = \begin{cases} pf_1 & \text{if } pf_2 \text{ is } \texttt{Id}, \\ pf_2 & \text{if } pf_1 \text{ is } \texttt{Id}, \\ pf_1.pf_2 & \text{otherwise.} \end{cases}$$

$$len(pf) = \begin{cases} 0 & \text{if } pf \text{ is } \texttt{Id}, \\ 1 + len(pf_1) & \text{otherwise, where } pf = pf_1 \circ P, \\ & \text{for some property } P. \end{cases}$$

Let X be a set of path functions $\{pf_1, \ldots, pf_n\}$. We write $pf \circ X$ to denote $\{pf \circ pf_1, \ldots, pf \circ pf_n\}$. □

Note that the composition operator is clearly associative; that is, $pf_1 \circ (pf_2 \circ pf_3) = (pf_1 \circ pf_2) \circ pf_3$. For example, with the UNIVERSITY schema, `S` ∘ `Name` is the path function `S.Name`, and both `Id` ∘ `C` and `C` ∘ `Id` are the path function `C`. The expression `Id` ∘ `C` ∘ `Room` denotes either (`Id` ∘ `C`)∘ `Room` or `Id` ∘(`C` ∘ `Room`), and in both cases is the path function `C.Room`. The following identity on *len* is also a straightforward consequence of our definitions.

$$len(pf_1 \circ pf_2) = len(pf_1) + len(pf_2)$$

Definition 4: A path $u \longrightarrow \cdots \longrightarrow w \xrightarrow{P} v$ in a database $G(V, A)$ for class schema S is *described* by a path function pf *iff* either (1) the path consists of a single vertex u and pf is `Id`, or (2) pf is $pf_1 \circ P$, where $u \longrightarrow \cdots \longrightarrow w$ is described by pf_1. □

For example, `In.Name` and `In.Head.In.Name` are path functions which describe paths from vertex u to v in Figure 3. Now consider that `Name.In` is also a path function according

to our definitions, but that no path can exist in any database for the UNIVERSITY schema which is described by `Name.In`. In [20], a subset of path functions for a given schema S, denoted $PF(S)$ below, is defined and proven to satisfy a completeness property for databases over S: any path in any database for S can be described by a path function in $PF(S)$, and any path function in $PF(S)$ describes a path in some database for S. The same reference also proves an important sense in which the composition operator remains closed over $PF(S)$. Both of the results are reproduced as Lemma 1 and Lemma 2 below.

Definition 5: The set of *well-formed path functions* $PF(S)$ over class schema S is the smallest set of path functions over S satisfying the following two conditions.

1. $\texttt{Id} \in PF(S)$, where
   (a) $Dom(\texttt{Id}) \stackrel{\text{def}}{=} Classes(S)$, and
   (b) $Ran(C, \texttt{Id}) \stackrel{\text{def}}{=} C$, for all $C \in Classes(S)$.
2. If $pf \in PF(S)$, $C \in Dom(pf)$ and $P \in Props(Ran(C, pf))$, then $pf \circ P \in PF(S)$, where
   (a) $Dom(pf \circ P) \stackrel{\text{def}}{=} \{C_1 \in Dom(pf) \mid P \in Props(Ran(C_1, pf))\}$, and
   (b) $Ran(C_1, pf \circ P) \stackrel{\text{def}}{=} Type(Ran(C_1, pf), P)$, for all $C_1 \in Dom(pf \circ P)$.

Capital letters X, Y and Z are used to denote finite subsets of $PF(S)$ for some class schema S, and XY, for example, denotes the union of path functions mentioned in X and Y. By a slight abuse of notation, we write $PathFuncs(C)$ to denote all path functions $pf \in PF(S)$ where $C \in Dom(pf)$, for $C \in Classes(S)$. A class schema S is *cyclic iff* there exists $pf \in PF(S) - \{\texttt{Id}\}$ and $C \in Dom(pf)$ where $C = Ran(C, pf)$. (A simple consequence is that S is cyclic *iff* $PF(S)$ is infinite.) □

Note that the subset of well-formed path functions for cyclic class schema, however, continues to be infinite. For example, the UNIVERSITY schema has a well-formed "head of the department" function `In.Head`, a "head of the department of the head of the department" function `In.Head.In.Head`, and so on. Other well-formed UNIVERSITY path functions include

`S`, `S.Name`, `C`, `C.Room`, `C.Time`, `C.Inst`, `C.Inst.In` and `C.Inst.In.Head`.

Note that each of these path functions is also in $PathFuncs(\texttt{enrollment})$. Also, for example,

$$Dom(\texttt{Name}) = \{\texttt{prof}, \texttt{dept}, \texttt{student}\}$$

and

$$Ran(\texttt{enrollment}, \texttt{C.Inst}) = \texttt{prof}.$$

Lemma 1: (*expressiveness of well-formed path functions—from* [20]) Let $G(V, A)$ be a database for a given class schema S. If a path $u \longrightarrow \cdots \longrightarrow v$ exists in G, then there exists a unique $pf' \in PathFuncs(l_{Cl}(u))$ describing $u \longrightarrow \cdots \longrightarrow v$. Also, for every $u \in V$ and $pf'' \in PathFuncs(l_{Cl}(u))$, there exists a path $u \longrightarrow \cdots \longrightarrow v$ in G described by pf''. □

Note that Lemma 1 also asserts that no two distinct paths with common end vertices can be described by the *same* path function (which motivates the use of the phrase "path function", as opposed to, say, "path description"). For example, vertex v in Figure 3 is the unique vertex reachable from vertex u by a path described by `In.Name`. By a slight abuse of notation, we write u.`In.Name` to denote v, and in general $u.pf$ to denote the unique vertex w reachable from u by a path described by pf, whenever $pf \in PathFuncs(l_{Cl}(u))$.

Lemma 2: (*closure of composition—also from* [20]) Assume $C \in Classes(S)$, for some class schema S. Then

$$pf_1 \in PathFuncs(C),\ pf_2 \in PF(S) \text{ and } Ran(C, pf_1) \in Dom(pf_2)$$

if and only if

$$pf_1 \circ pf_2 \in PathFuncs(C).$$ □

The remaining definitions in this section present the syntax of our functional constraint language, and define satisfaction and logical consequence as they relate to the above graph-based view of databases.

Definition 6: The syntax of a *path functional dependency* (PFD) over a class schema S is given by

$$C(pf_1 \cdots pf_m \rightarrow pf_{m+1} \cdots pf_n).$$

Such a constraint is *well-formed* if (1) $0 \leq m < n$ and (2) $pf_i \in PathFuncs(C)$ for $1 \leq i \leq n$. (This definition differs slightly from the one given in [20]—we now admit PFDs with no path functions occurring before the arrow.)

A *key path functional dependency* (key PFD) satisfies the condition that, for any pf_j where $m < j \leq n$, there exists a path function $pf_j{}'$ such that $pf_j \circ pf_j{}' = pf_i$ for some $1 \leq i \leq m$; that is, that every right-hand side path function is a "prefix" of some left-hand side path function. We say that a key PFD is *simple* if $n = m + 1$ and pf_n is the path function `Id`; that is, if the single path function `Id` occurs on the right-hand side. (This definition also differs from the one given in [20]. The notion of a key PFD has been somewhat generalized to include what we have called complex or embedded keys in our introductory comments.)

$C(pf_1 \cdots pf_m \rightarrow pf_{m+1} \cdots pf_n)$ is *satisfied* by a database $G(V, A)$ for S *iff* for any pair of vertices $u, v \in V$ where $l_{Cl}(u) = l_{Cl}(v) = C$, $u.pf_i = v.pf_i$ for all $1 \leq i \leq m$ implies $u.pf_j = v.pf_j$ for all $m < j \leq n$. Note that the antecedent is trivially satisfied when $m = 0$. In this case, $u.pf_j = v.pf_j$ for all $1 \leq j \leq n$ unconditionally.[1] □

[1] For example, a PFD of the form $C(\emptyset \rightarrow$ `Id`$)$ is only satisfied by a database with at most one C object.

By Lemma 1, any PFD that is not well-formed is always (trivially) satisfied. Also note that a schema graph, such as the one depicted in Figure 1 for the UNIVERSITY schema, must satisfy *any* well-formed PFD when viewed as a database since no class has more than one object. In contrast, the UNIVERSITY database in Figure 3 illustrates a violation of the first key PFD in Table 1 on class `dept` (two distinct department objects have the same name).

Definition 7: (*logical consequence*) Let F denote a finite set of PFDs over a class schema S, and let f denote an arbitrary PFD also over S. Then f is a *logical consequence* of F, written $F \models_S f$, *iff* any database $G(V, A)$ satisfying all constraints in F must also satisfy f. If S is clear from the context, then we write $F \models f$. □

3. On Proof Theories for PFD Constraints

3.1 A Complete Axiomatization for the Implication Problem

In [20], it was proven that the inference axioms for PFDs listed in Table 3 are sound, and that axioms A1 to A5 are complete if there are no PFDs of the form $C(\emptyset \rightarrow X)$ in F. In this subsection, we extend this earlier work to show that allowing PFDs with empty left-hand sides does not alter the theory; that is, that axioms A1 to A5 in Table 3 remain complete. A proof theory based on the axioms is given as follows.

Definition 8: Let $F \cup \{C(X \rightarrow Y)\}$ denote a finite set of PFDs over class schema S. There is a derivation of $C(X \rightarrow Y)$ from F, written $F \vdash C(X \rightarrow Y)$, *iff* the former is a member of F, or can be derived from F with the use of any of the inference axioms in Table 3. A PFD $C(X \rightarrow Y)$ over S is *trivial iff* $\emptyset \models C(X \rightarrow Y)$. Also, if $X \subseteq \mathit{PathFuncs}(C)$, for some class scheme C, then X^+ denotes the smallest set containing all $pf \in \mathit{PathFuncs}(C)$ where $F \vdash C(X \rightarrow pf)$. (Note that X^+ may not be finite.) □

Both the earlier proof of completeness in [20] and our modification to the proof require the construction and manipulation of a special kind of database called a *C-Tree*.

Definition 9: Let C denote an arbitrary class in $\mathit{Classes}(S)$, for some schema S. A *C-Tree* is a database $G(V, A)$ of S constructed in two steps as follows.

Step 1. For each $pf \in \mathit{PathFuncs}(C)$, add vertex u with $l_{Cl}(u)$ assigned $\mathit{Ran}(C, pf)$, and with an additional vertex label $l_{Pf}(u)$ (called its *path function labeling*) assigned pf. The single vertex v with $l_{Pf}(v) = \mathtt{Id}$ is denoted as $\mathit{Root}(v)$.

Step 2. For each $u, v \in V$ where $l_{Pf}(u) = pf$ and $l_{Pf}(v) = pf \circ P$, add $u \xrightarrow{P} v$ to A.

A *partial* C-Tree is a subtree of a C-Tree with the same root. (A partial C-Tree may be a C-Tree as a special case.) For any vertex u in a partial C-Tree, we refer to $\mathit{len}(l_{Pf}(u))$ as the *depth* of u. □

Table 3: Axioms for PFDs.

name	*definition*
A1 (*reflexivity*)	$\dfrac{\emptyset \subset Y \subseteq X \subseteq PathFuncs(C)}{C(X \rightarrow Y)}$
A2 (*augmentation*)	$\dfrac{C(X \rightarrow Y),\ Z \subseteq Pathfuncs(C)}{C(XZ \rightarrow YZ)}$
A3 (*transitivity*)	$\dfrac{C(X \rightarrow Y),\ C(Y \rightarrow Z)}{C(X \rightarrow Z)}$
A4 (*simple attribution*)	$\dfrac{P \in Props(C)}{C(\texttt{Id} \rightarrow P)}$
A5 (*simple substitution*)	$\dfrac{P \in Props(C),\ Type(C,P)(X \rightarrow Y)}{C(P \circ X \rightarrow P \circ Y)}$
A6 (*additivity*)	$\dfrac{C(X \rightarrow Y), C(X \rightarrow Z)}{C(X \rightarrow YZ)}$
A7 (*projectivity*)	$\dfrac{C(X \rightarrow YZ)}{C(X \rightarrow Y)}$
A8 (*attribution*)	$\dfrac{pf \in PathFuncs(C)}{C(\texttt{Id} \rightarrow pf)}$
A9 (*substitution*)	$\dfrac{pf \in PathFuncs(C),\ Ran(C,pf)(X \rightarrow Y)}{C(pf \circ X \rightarrow pf \circ Y)}$

An example partial `course`-Tree for the UNIVERSITY schema appears in Figure 4. Note that we have indicated the additional path function labeling for each vertex in parenthesis below the class labeling. Also note that, although this tree is finite, a full `course`-Tree would necessarily be infinite since $PathFuncs(\texttt{course})$ is infinite.

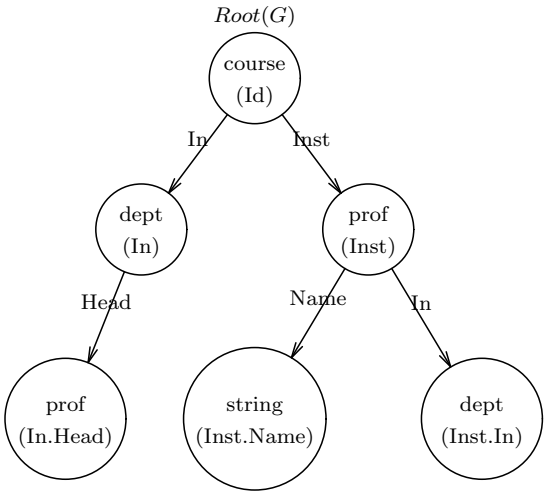

Figure 4: A PARTIAL `course`-TREE FOR THE UNIVERSITY SCHEMA.

The properties satisfied by a C-Tree, which are important to our presentation, are given in the following lemma.

Lemma 3: Let $G(V, A)$ be a partial C-Tree over schema S, where $C \in Classes(S)$. Then the following three conditions hold:

C1: For every $u \in V$ and every $pf \in PathFuncs(l_{Cl}(u))$, if $u.pf \in V$, then $l_{Pf}(u) \circ pf = l_{Pf}(u.pf)$.

C2: For every $u \in V$, $u = Root(G).l_{Pf}(u)$.

C3: If $G(V, A)$ is a *full* C-Tree, then for every $pf \in PathFuncs(C)$, there is a unique vertex $u \in V$ such that $u = Root(G).pf$.

Proof. (See proof of Lemma 7 in [20].) □

A simple consequence of condition C2 is that the depth of any vertex in a C-Tree G is its path length from $Root(G)$. For example, the depth of the single `string` vertex in the partial `course`-Tree of Figure 4 is $len(\texttt{Inst.Name})$ $(= 2)$.

Now let F be a finite set of PFDs over S which contains PFDs with empty left-hand sides. Along the same line as proving Theorem 2 in [20], it can be shown that inference axioms $A1$ to $A5$ are sound; that is, $F \vdash C(X \rightarrow Y)$ implies $F \models C(X \rightarrow Y)$ for any PFD $C(X \rightarrow Y)$ over S. Hence inference axioms $A6$ to $A9$ are also sound by Lemma 5 in [20].

In general, to prove that inference axioms $A1$ to $A5$ are complete, it suffices to show that $F \nvdash C(X \rightarrow Y)$ implies $F \not\models C(X \rightarrow Y)$; that is, if $Y \not\subseteq X^+$, then to construct a database for S that satisfies F but not $C(X \rightarrow Y)$. We may also assume, without loss of generality,

that no PFD in F is trivial, and furthermore, by additivity $A6$ and projectivity $A7$, that the right-hand side of every PFD in F consists of a single path function; that is, that every PFD in F is of the form $C(Z \rightarrow pf)$.[2] The earlier completeness proof in [20] constructed such a database, called a *Two-C-Tree*, from two copies of a (complete) C-tree. The important conditions satisfied by a Two-C-Tree database G are as follows.

1. G contains two vertices $R1$ and $R2$ in which $l_{Cl}(R1) = l_{Cl}(R2) = C$ and such that $R1.pf = R2.pf$ *iff* $pf \in X^+$ for every $pf \in PathFuncs(C)$. (Thus, if $Y \not\subseteq X^+$, then G must fail to satisfy $C(X \rightarrow Y)$.)

2. G satisfies F (provided that no PFD in F has an empty left-hand side).

The main difficulty with a Two-C-Tree $G(V, A)$, if F has PFD constraints with empty left-hand sides, is that G might contain distinct vertices $u, v \in V$ in which $l_{Cl}(u) = l_{Cl}(v) = C'$, for some $C' \in Classes(S)$, and for which $u = R1.pf_1 = R2.pf_1$ and $v = R1.pf_2 = R2.pf_2$, for some $pf_1, pf_2 \in PathFuncs(C)$ (i.e. $pf_1, pf_2 \in X^+$). Then, for example, G will fail to satisfy F should it contain the constraint $C'(\emptyset \rightarrow \mathtt{Id})$.[3]

Roughly, our refinement to the earlier proof is based on a modification to the definition of a Two-C-Tree which overcomes this problem. The modification, call a *Two-C-Graph*, will satisfy the condition that, for each $C' \in Classes(S)$, there is a *unique* vertex v such that $R1.pf = R2.pf = v$ for every $pf \in PathFuncs(C)$ with $pf \in X^+$ and $Ran(C, pf) = C'$. We prove that a Two-C-Graph database will satisfy all PFDs in F, including any with empty left-hand sides.

As in the earlier case of a Two-C-Tree, the construction of a Two-C-Graph starts with two copies of a (complete) C-Tree. However, unlike the previous case, another special kind of database is also used. In our introductory comments, we referred to such a database as a *schema graph*.

Definition 10: A *schema graph* for S is a directed graph $G_s(V_s, A_s)$ satisfying the following two conditions. (Note that $G_s(V_s, A_s)$ is clearly a database for S.)

SG1: For each $C \in Classes(S)$, there is a unique vertex $v \in V_s$ such that $l_{Cl}(v) = C$.

SG2: For each $C \in Classes(S)$ and each $P \in Props(C)$, there is a unique arc $u \xrightarrow{P} v \in A_s$ such that $l_{Cl}(u) = C$ and $l_{Cl}(v) = Type(C, P)$. □

For example, the database illustrated in Figure 1 is a schema graph for the UNIVERSITY class schema listed in Table 2. Another example of a class schema and corresponding schema graph appears in Figure 5.

Finally, the definition of a Two-C-Graph will rely on the following "suffix closure" condition for X^+.

[2] No PFD in F has an empty *right*-hand side, since, by the reflexivity axiom $A1$, such a PFD would be trivial.

[3] The problem generalizes to any PFD constraint of the form $C'(\emptyset \rightarrow pf)$.

```
a{ B:b, C:c }
b{ A:a, D:d }
c{ G:b, E:e }
d{ F:f }
e{ }
f{ }
```

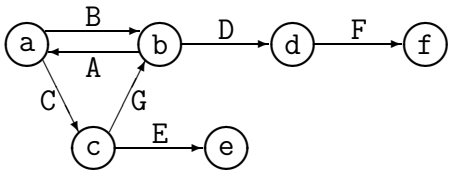

Figure 5: A CLASS SCHEMA AND ITS SCHEMA GRAPH.

Lemma 4: For $pf \in PathFuncs(C)$, if $pf \in X^+$, then $pf \circ pf' \in X^+$ for every $pf \circ pf' \in PathFuncs(C)$.

Proof. Let $C' = Ran(C, pf)$. Then $pf \circ pf' \in PathFuncs(C)$ implies $pf' \in PathFuncs(C')$, by Lemma 2. By attribution $A8$, we can derive $C'(\mathtt{Id} \rightarrow pf')$, and thus $C(pf \rightarrow pf \circ pf')$ by substitution $A9$. Since $pf \in X^+$ implies $F \vdash C(X \rightarrow pf)$, $C(X \rightarrow pf)$ and $C(pf \rightarrow pf \circ pf')$ imply $C(X \rightarrow pf \circ pf')$ by transitivity $A3$. Hence $pf \circ pf' \in X^+$. □

Now consider where there is a PFD $C(X \rightarrow Y)$ such that $F \nvdash C(X \rightarrow Y)$; that is, $Y \not\subseteq X^+$. Then, since $pf = \mathtt{Id} \circ pf$ for any $pf \in PathFuncs(C)$, it follows from Lemma 4 that

$$\mathtt{Id} \notin X^+. \tag{3.1}$$

Definition 11: Let $F \cup \{C(X \rightarrow Y)\}$ denote a set of PFDs such that $F \nvdash C(X \rightarrow Y)$. A *Two-C-Graph* is a (possibly infinite) directed graph $G(V, A)$ constructed as follows.

Step 1. Construct two C-Trees $G_1(V_1, A_1)$ and $G_2(V_2, A_2)$, and a schema graph $G_s(V_s, A_s)$. $Root(G_1)$ and $Root(G_2)$ are denoted by $R1$ and $R2$, respectively.

Step 2. Let $V_i' = \{v \in V_i \mid l_{Pf}(v) \notin X^+\}$ for $i = 1, 2$. Note that $R1 \in V_1'$ and $R2 \in V_2'$ by (3.1). Let $A_i' = \{u \xrightarrow{P} v \in A_i \mid u, v \in V_i'\}$ for $i = 1, 2$. Add all vertices in $V_s \cup V_1' \cup V_2'$ to V and all arcs in $A_s \cup A_1' \cup A_2'$ to A.

Step 3. For each $u \in V_1' \cup V_2'$ and each $P \in Props(l_{Cl}(u))$ where $u \xrightarrow{P} v \notin A$ for any $v \in V$, add an arc $u \xrightarrow{P} w$ to A, where $w \in V_s$ and $l_{Cl}(w) = Type(l_{Cl}(u), P)$. Note that w is unique by condition SG1 of Definition 10. □

From Lemma 4, it should be clear that a Two-C-Graph is symmetric with respect to $R1$ and $R2$. An outline of the form of a Two-C-Graph is illustrated in Figure 6, in which we denote a (possibly infinite) partial C-Tree consisting of V_i' and A_i' by the label $G_i'(V_i', A_i')$, where $i = 1, 2$. The arcs added in Step 3 are also indicated.

Lemma 5: The Two-C-Graph $G(V, A)$ is a database for S satisfying the following three conditions.

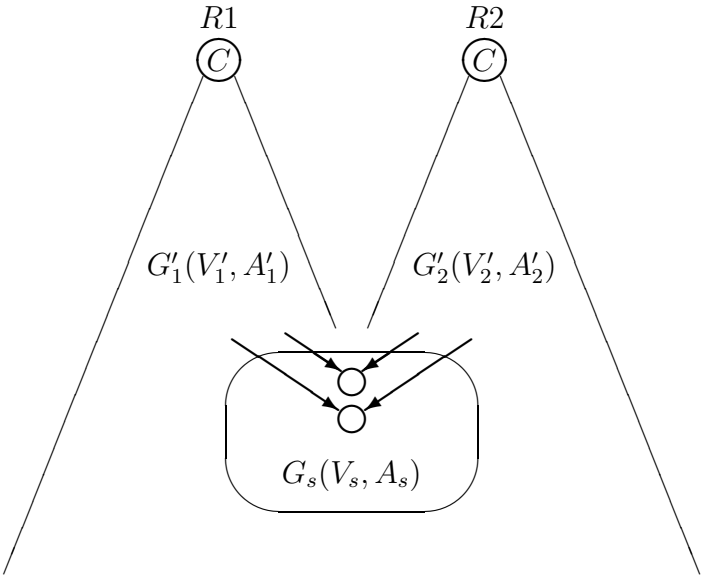

Figure 6: General form of a *Two-C-Graph*.

TCG1: (a) If $u \in V_s$, then $u.pf \in V_s$ for every $pf \in PathFuncs(l_{Cl}(u))$.

(b) If $u \in V'_i$, then exactly one of $u.pf \in V'_i$ and $u.pf \in V_s$ holds for every $pf \in PathFuncs(l_{Cl}(u))$, where $i = 1$ or 2.

TCG2: For every $pf \in PathFuncs(C)$, $pf \in X^+$ *iff* $R1.pf \in V_s$ *iff* $R1.pf = R2.pf$.

TCG3: For any pair of distinct vertices $u, v \in V$ where $l_{Cl}(u) = l_{Cl}(v)$, if $u.pf = v.pf$ for some $pf \in PathFuncs(l_{Cl}(u))$, then $u.pf \in V_s$.

Proof. Since $G_s(V_s, A_s)$ is a database for S, it can be proven with the same line of argument used in the proof of Lemma 8 in [20] that $G(V, A)$ is also a database for S.

Now consider TCG1. Clearly, just after Step 2 in Definition 11, for every $u \xrightarrow{P} v \in A$, exactly one of $\{u, v\} \subseteq V'_1$, $\{u, v\} \subseteq V'_2$, and $\{u, v\} \subseteq V_s$ holds. Furthermore, for each arc $u \xrightarrow{P} v$ added to A in Step 3, $u \in V'_1 \cup V'_2$ and $v \in V_s$. Thus, after Step 3, for every $u \xrightarrow{P} v \in A$, exactly one of $v \in V_s$, $\{u, v\} \subseteq V'_1$, and $\{u, v\} \subseteq V'_2$ must be true. TCG1(a) therefore follows. Since $V'_i \cap V_s = \emptyset$, TCG1(b) also follows.

Consider TCG2. We first prove that $pf \in X^+$ *iff* $R1.pf \in V_s$. Since $R1 \in V'_1$, it follows from TCG1(b) that $R1.pf \in V_s$ *iff* $R1.pf \notin V'_1$. Thus, it suffices to show that

$$pf \notin X^+ \text{ iff } R1.pf \in V'_1. \tag{3.2}$$

It follows from the definition of V'_1 that, for every $v \in V_1$, $l_{Pf}(v) \notin X^+$ *iff* $v \in V'_1$. This implies (3.2) since there is a one-to-one correspondence between V_1 and $PathFuncs(C)$ according to conditions C2 and C3 of Lemma 3. We next prove that $R1.pf \in V_s$ *iff* $R1.pf = R2.pf$. Assume $R1.pf \in V_s$. Since the Two-C-Graph is symmetric with respect to $R1$ and $R2$, $R1.pf \in V_s$ implies both $R2.pf \in V_s$ and $l_{Cl}(R1.pf) = l_{Cl}(R2.pf)$. Thus $R1.pf = R2.pf$ by condition SG1

of Definition 10. Conversely, since $R1 \in V_1'$, $R2 \in V_2'$ and $V_1' \cap V_2' = \emptyset$, it follows from TCG1(b) that $R1.pf = R2.pf$ implies $R1.pf \in V_s$.

Finally consider TCG3. Assume that $u.pf = v.pf$ but that $u.pf \notin V_s$ for two distinct vertices $u, v \in V$. By TCG1(a), $u.pf \notin V_s$ implies $u \notin V_s$, that is, $u \in V_1' \cup V_2'$. Assume without loss of generality that $u \in V_1'$. By TCG1(b), $u \in V_1'$ and $u.pf \notin V_s$ imply $u.pf (= v.pf) \in V_1'$. By TCG1(a) and (b), $v.pf \in V_1'$ implies $v \in V_1'$; that is, the three vertices u, v, and $u.pf$ $(= v.pf)$ are in V_1'. Since $G_1'(V_1', A_1')$ is a partial C-Tree, it follows from condition C2 of Lemma 3 that

$$w = R1.l_{Pf}(w) \text{ for every } w \in V_1'. \tag{3.3}$$

Furthermore by condition C1 of that lemma, $u.pf = v.pf$ implies $l_{Pf}(u) \circ pf = l_{Pf}(v) \circ pf$, that is, $l_{Pf}(u) = l_{Pf}(v)$, and therefore that $u = v$—a contradiction with our assumption above that u and v are distinct vertices. Hence, if $u \neq v$ and $u.pf = v.pf$, then $u.pf \in V_s$. □

Lemma 6: For $pf \in PathFuncs(C)$, if there is a PFD $C'(Z \rightarrow pf') \in F$ such that $C' = Ran(C, pf)$ and $pf \circ Z \subseteq X^+$, then $pf \circ pf' \in X^+$.

Proof. Since $C' = Ran(C, pf)$, $C'(Z \rightarrow pf')$ implies $C(pf \circ Z \rightarrow pf \circ pf')$ by substitution $A9$. Since $pf \circ Z \subseteq X^+$, $F \vdash C(X \rightarrow pf \circ Z)$ by definition. Hence $C(X \rightarrow pf \circ Z)$ and $C(pf \circ Z \rightarrow pf \circ pf')$ imply $C(X \rightarrow pf \circ pf')$ by transitivity $A3$. That is, $pf \circ pf' \in X^+$. □

Theorem 1: Inference axioms $A1$ to $A5$ are sound and complete, even in the case that there are PFDs with empty left-hand sides.

Proof. It suffices to prove that a Two-C-Graph $G(V, A)$ is a database satisfying F but not $C(X \rightarrow Y)$.

We first show that $G(V, A)$ does not satisfy $C(X \rightarrow Y)$. Since $X \subseteq X^+$ by reflexivity $A1$, it follows from condition TCG2 of Lemma 5 that $R1.pf = R2.pf$ for every $pf \in X$. Conversely, since $Y \not\subseteq X^+$, it follows from condition TCG2 that $R1.pf \neq R2.pf$ for some $pf \in Y$. Hence $G(V, A)$ does not satisfy $C(X \rightarrow Y)$.

We next show that $G(V, A)$ satisfies F. Assume that $G(V, A)$ does not satisfy a PFD $C'(Z \rightarrow pf) \in F$. Then there are two distinct vertices $u, v \in V$ such that

$$l_{Cl}(u) = l_{Cl}(v) = C', \tag{3.4}$$

$$u.pf_z = v.pf_z \text{ for every } pf_z \in Z,^4 \text{ and} \tag{3.5}$$

$$u.pf \neq v.pf. \tag{3.6}$$

Since $u.pf \notin V_s$ or $v.pf \notin V_s$ by (3.4), (3.6), and condition SG1 of Definition 10, assume without loss of generality that

$$u.pf \in V_1'. \tag{3.7}$$

Then $u \in V_1'$ by conditions TCG1(a) and (b) of Lemma 5. It follows from property functionality and property value completeness for the database $G(V, A)$ that for every $w \in V$ and

[4] If $Z = \emptyset$, then this condition holds trivially.

every $pf' \in PathFuncs(l_{Cl}(w))$, there is a unique vertex $w' \in V$ such that $w' = w.pf'$. Since $u = R1.l_{Pf}(u)$ by $u \in V_1'$ and (3.3) in the proof of Lemma 5,

$$u.pf' = R1.l_{Pf}(u) \circ pf' \text{ for every } pf' \in PathFuncs(l_{Cl}(u)). \tag{3.8}$$

And since $u.pf_z \in V_s$ by (3.5) and condition TCG3 of Lemma 5, $R1.l_{Pf}(u) \circ pf_z \in V_s$ for every $pf_z \in Z$ by (3.8). Thus, by condition TCG2 of Lemma 5,

$$l_{Pf}(u) \circ Z \subseteq X^+. \tag{3.9}$$

Since $Ran(C, l_{Pf}(u)) = l_{Cl}(u) = C'$ by (3.4) and $C'(Z \rightarrow pf)$ is in F, it follows from Lemma 6 and (3.9) that $l_{Pf}(u) \circ pf \in X^+$. Conversely, since $R1.l_{Pf}(u) \circ pf \in V_1'$ by (3.7) and (3.8), it follows from the definition of V_1' that $l_{Pf}(u) \circ pf \notin X^+$—a contradiction. Therefore, $G(V, A)$ satisfies F. □

3.2 The Inequivalence of the Finite Implication Problem

In this subsection, we prove that the inference axioms for PFDs listed in Table 3 are *not* complete if databases with infinitely many objects are disallowed. In particular, we exhibit a class schema S and finite set $F \cup \{C(X \rightarrow Y)\}$ of PFDs over S such that $F \not\models C(X \rightarrow Y)$, but in which $C(X \rightarrow Y)$ is necessarily satisfied by any *finite* database for S that satisfies F. If this were not the case, if the implication problem and finite implication problem for PFDs were equivalent, then the existence of the semi-decision procedure for PFDs in [20] would immediately imply the decidability of both problems [6].[5]

Definition 12: Let $F \cup \{C(X \rightarrow Y)\}$ denote a finite set of PFDs over a given class schema S. $C(X \rightarrow Y)$ is a *finite* logical consequence of F, written $F \models_{\text{finite}} C(X \rightarrow Y)$, *iff* any *finite* database for S satisfying F must also satisfy $C(X \rightarrow Y)$. □

Lemma 7: Let S consist of the following two complex object types.

```
a{ A:a, B:b }
b{ }
```

Then `a( A.B → Id )` $\not\models$ `a( B → Id )` and `a( A.B → Id )` $\models_{\text{finite}}$ `a( B → Id )`.

Proof. Since `a( A.B → Id )` is a simple key PFD, the closure of `B` can be computed efficiently by Theorem 6 in Section 6. In fact, it is easy to verify that $\texttt{B}^+ = \{\ \texttt{B}\ \}$. Thus: `a( A.B → Id )` $\not\models$ `a( B → Id )`.

Assume that `a( A.B → Id )` $\not\models_{\text{finite}}$ `a( B → Id )`. By definition, there must exist a finite database $G(V, A)$ for S that satisfies `a( A.B → Id )` but does not satisfy `a( B → Id )`. Then $G(V, A)$ contains a subgraph given in Figure 7, where v_1 and v_2 are distinct. There are two cases to be considered.

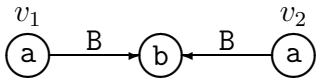

Figure 7: A SUBGRAPH OF $G(V, A)$.

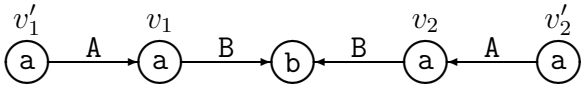

Figure 8: A SUBGRAPH OF $G(V, A)$ IN CASE 1.

Case 1. Consider where both v_1 and v_2 have in-arcs labeled "A"; that is, $G(V, A)$ contains a subgraph given in Figure 8. Since v_1 and v_2 are distinct, v'_1 and v'_2 are also distinct by property functionality. However, this implies that $G(V, A)$ could not satisfy a(A.B → Id), no matter how the other part of $G(V, A)$ would be constructed—a contradiction. Note that v_i may coincide with v'_j, unless $v'_1 = v'_2$, where $i = 1$ or 2, and $j = 1$ or 2.

Case 2. Now consider where either v_1 has no in-arc labeled "A" or v_2 has no in-arc labeled "A". Assume without loss of generality that v_1 has no in-arc labeled "A". By property value completeness, there is an arc $v_1 \xrightarrow{\texttt{A}} u_1 \in A$ where $l_{Cl}(u_1) = \mathit{Type}(l_{Cl}(v_1), \texttt{A}) = \mathit{Type}(\texttt{a}, \texttt{A}) = \texttt{a} = l_{Cl}(v_1)$. By a similar argument, there is an arc $u_1 \xrightarrow{\texttt{A}} u_2 \in A$, where $l_{Cl}(u_2) = \texttt{a}$. Therefore, in general, $G(V, A)$ contains an infinite sequence $v_1 \xrightarrow{\texttt{A}} u_1 \xrightarrow{\texttt{A}} u_2 \xrightarrow{\texttt{A}} \cdots$ in which all vertices are labeled "a". Now, since $G(V, A)$ is finite, at least one vertex occurs infinitely in the sequence. Furthermore, v_1 has no in-arc labeled "A" by assumption. Thus, the sequence contains a subsequence $v_1 \xrightarrow{\texttt{A}} u_1 \xrightarrow{\texttt{A}} \cdots \xrightarrow{\texttt{A}} u_i \xrightarrow{\texttt{A}} \cdots \xrightarrow{\texttt{A}} u_i$ in which u_i occurs twice but all other vertices occur at most once. This implies that the two in-arcs of u_i must be distinct. Let $w \xrightarrow{\texttt{A}} u_i$ and $w' \xrightarrow{\texttt{A}} u_i$ be the two distinct in-arcs of u_i. Since (1) w and w' are distinct by property functionality and (2) u_i has an out-arc labeled "B" by property value completeness, $G(V, A)$ contains a subgraph given in Figure 9. However, this demonstrates that $G(V, A)$ can never satisfy a(A.B → Id), no matter how the other part of $G(V, A)$ might be constructed—a contradiction. (Note that u_i may coincide with w or w', unless $w = w'$.)

Therefore, there is no finite database that satisfies a(A.B → Id) but does not satisfy a(B → Id). That is, a(A.B → Id) $\models_{\text{finite}}$ a(B → Id). □

[5]This justifies our use of infinite databases in our decidability proof given the next section.

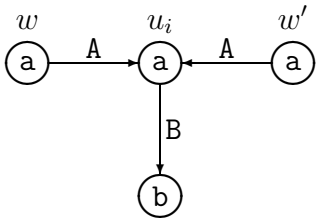

Figure 9: A SUBGRAPH OF $G(V, A)$ IN CASE 2.

By Lemma 7, we have the following theorem.

Theorem 2: Finite logical implication for PFDs is different from (infinite) logical implication for PFDs, even if all given PFDs are simple key PFDs. □

Theorem 2 implies that inference axioms $A1$ to $A5$, although sound, are no longer complete for finite logical implication for PFDs.

4. Decidability of the Implication Problem for Arbitrary PFDs

We now resolve an important open issue concerning PFD theory: we prove that the implication problem for arbitrary PFDs is decidable. Our proof is based on an earlier semi-decision procedure for the problem, given in [20], which returns "yes" if $F \models C(X \rightarrow Y)$ (and, of course, might not terminate if $F \not\models C(X \rightarrow Y)$). Although the following revision to this procedure is less effective (it may also not terminate if $F \models C(X \rightarrow Y)$), much of its functionality has been factored into another procedure, called MARK, which will be essential to our presentation. Procedure MARK assumes that each vertex v in a database can be assigned an additional boolean valued *mark* label, denoted $Mark(v)$. In the discussion following, we refer to a vertex v as *marked* (resp. *unmarked*) if $Mark(v)$ has the value *true* (resp. *false*).

Procedure 1:
Input: a finite set $F \cup \{C(X \rightarrow Y)\}$ of PFDs over S.
Output: "yes" *iff* $F \models C(X \rightarrow Y)$.
Method: For a C-Tree $G_c(V_c, A_c)$, execute procedure MARK(G_c, X) (defined immediately following). If $Root(G_c).pf$ is marked for every $pf \in Y$, then output "yes"; otherwise, output "no".

procedure MARK(G, X)
Input: a partial C-Tree $G(V, A)$ and a finite subset X of $PathFuncs(C)$.

Step 1. For each $v \in V$, assign $Mark(v)$ the value *false*.

Step 2. For each $pf \in X$, assign $Mark(Root(G).pf)$ the value *true*.

Step 3. Apply the following two rules to $G(V, A)$ exhaustively.

> **Rule 1:** If vertex v has an ancestor u which is marked, then assign $Mark(v)$ the value *true*.
>
> **Rule 2:** If u is a vertex and $C'(Z \rightarrow pf)$ a PFD in F such that (1) $C' = l_{Cl}(u)$, and (2) $Mark(u.pf_z)$ is *true* for every $pf_z \in Z$,[6] then assign $Mark(u.pf)$ the value *true*.

For an example of running the MARK procedure, recall the class schema appearing in Figure 5, and assume F consists of the following four PFDs.

[6] If $Z = \emptyset$, then condition (2) holds trivially.

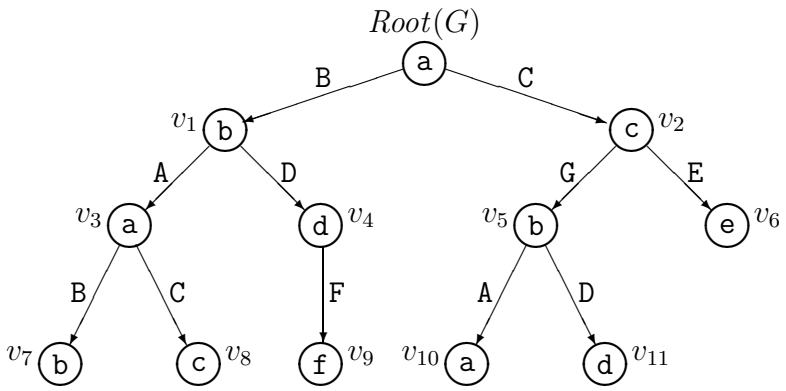

Figure 10: A PARTIAL "a"-TREE $G(V, A)$.

f_1: a(B.D C.G → Id)
f_2: a(B.D.F C.G.D → C.E)
f_3: b(D.F → A)
f_4: d(F → Id)

Also assume the call to procedure MARK is of the form

MARK(G, { B.D, C.G.D }),

where G is the partial "a"-Tree given in Figure 10. At the end of Step 2, vertices v_4 and v_{11} are marked and all other vertices are unmarked. Now consider what happens during Step 3. Since vertex v_9 has an ancestor v_4 which is marked, Rule 1 updates G by assigning $Mark(v_9)$ the value *true*. The conditions for two applications of Rule 2 are then satisfied. The first case relates to vertex $Root(G)$ and PFD f_2 since (1) $l_{Cl}(Root(G)) =$ a and (2) both $Root(G)$.B.D.F ($= v_9$) and $Root(G)$.C.G.D ($= v_{11}$) are marked. The second case relates to vertex v_1 and PFD f_3 since (1) $l_{Cl}(v_1) =$ b and (2) v_1.D.F ($= v_9$) is marked. Rule 2 will therefore update G by assigning $Mark(Root(G)$.C.E) and $Mark(v_1$.A) the value *true*. Thus, vertices v_6 and v_3, which were previously unmarked, are now marked. Finally, since v_7 and v_8 are vertices with the marked ancestor v_3, Step 1 again updates G by assigning $Mark(v_7)$ and $Mark(v_8)$ the value *true*.

At this point, observe that no unmarked vertex in V can be changed to a marked status by applying Rule 1 to any vertex in V, or by applying Rule 2 to any vertex in V and any PFD in F. For example, although vertex v_4 and PFD f_4 satisfy the preconditions for a "firing" of Rule 2, since (1) $l_{Cl}(v_4) =$ d and (2) v_4.F ($= v_9$) is marked, the firing does not change G since it will not affect the marked status of v_4.Id ($= v_4$) (which is already marked). Also, since there is no chance to apply Rule 2 to PFD f_1 and any vertex in V, we may assume the original call to procedure MARK will terminate with the result that only vertices v_3, v_4, v_6, v_7, v_8, v_9, and v_{11} are marked.

Note that once a vertex satisfies the preconditions for either Rule 1 or Rule 2 in Step 3, then it will continue to satisfy the same preconditions throughout the remaining execution of Step 3. This holds since neither rule updates G by changing the status of a vertex from one

that is marked to one that is unmarked. Thus, the final selection of marked vertices in G after any call to MARK will not depend on the order of application of Rules 1 and 2 in Step 3.

The next lemma relates the path function labeling of a marked vertex, following a call to procedure MARK, to its membership in an important closure.

Lemma 8: Assume that MARK(G, X) is executed for a partial C-tree $G(V, A)$. For every $pf \in \mathit{PathFuncs}(C)$, if $\mathit{Root}(G).pf$ exists and is marked, then $pf \in X^+$.

Proof. By induction on the sequence of applications of Rules 1 and 2 in Step 3.
Basis. Since $X \subseteq X^+$ by reflexivity $A1$, for each $\mathit{Root}(G).pf$ which is marked at the end of Step 2, pf is in X^+.
Induction. Consider the nth application of either Rule 1 or Rule 2 in Step 3 which results in changing a vertex v in G from an unmarked state to a marked state. If Rule 1 applies, then v has an ancestor u which is marked. Condition C2 of Lemma 3 implies that $u = \mathit{Root}(G).l_{Pf}(u)$ and $v = \mathit{Root}(G).l_{Pf}(v)$. Since v is a descendant of u, there is a path function $pf \in \mathit{PathFuncs}(l_{Cl}(u))$ such that $u.pf = v$. Thus, by condition C1 of Lemma 3, $l_{Pf}(v) = l_{Pf}(u) \circ pf$. Since u is marked, the induction hypothesis implies $l_{Pf}(u)$ is in X^+. Therefore, by Lemma 4, $l_{Pf}(u) \circ pf$ $(= l_{Pf}(v))$ is in X^+, and the lemma follows since $v = \mathit{Root}(G).l_{Pf}(v)$.

Now consider where Rule 2 applies. Then there is a PFD $C'(Z \rightarrow pf) \in F$ and a (not necessarily proper) ancestor u in G such that (1) $C' = l_{Cl}(u)$, (2) $u.pf_z$ is marked for every $pf_z \in Z$, and (3) $u.pf = v$. Since $u = \mathit{Root}(G).l_{Pf}(u)$ by condition C2 of Lemma 3, condition (2) implies $l_{Pf}(u) \circ Z \subseteq X^+$ by the induction hypothesis. It then follows from Lemma 6 and condition (1) that $l_{Pf}(u) \circ pf \in X^+$. Since $v = \mathit{Root}(G).l_{Pf}(u) \circ pf$ by conditions C1 and C2 of Lemma 3, the lemma again follows. □

The correctness of Procedure 1 is now a simple consequence of the following lemma (since $F \models C(X \rightarrow Y)$ *iff* $Y \subseteq X^+$).

Lemma 9: Let $G'_c(V'_c, A'_c)$ be the state of a C-Tree G_c after a call of the form

$$\text{“MARK}(G_c(V_c, A_c), X)\text{”},$$

where $X \subseteq \mathit{PathFuncs}(C)$, and let *Marked* denote the set

$$\{l_{Pf}(v) \mid v \in V'_c \text{ and } \mathit{Mark}(v)\}.$$

Then: $\mathit{Marked} = X^+$.

Proof. Assume $\mathit{Marked} \neq X^+$. Since $\mathit{Marked} \subseteq X^+$ by Lemma 8, the assumption implies that $\mathit{Marked} \subset X^+$ ($A \subset B$ means that A is a proper subset of B). By Theorem 5 in [20], there must exist at least one PFD $C(Z \rightarrow pf) \in F_1(C) \cup F_2(C)$ such that $Z \subseteq \mathit{Marked}$ and $pf \notin \mathit{Marked}$. Here, $F_1(C)$ is the set of PFDs of the form “$C(pf' \rightarrow pf' \circ P)$”, where $pf', pf' \circ P \in \mathit{PathFuncs}(C)$, and $F_2(C)$ is the set of PFDs that can be derived from F by a single use of substitution $A9$. Hense, there are two cases to consider.

Case 1: where $C(Z \rightarrow pf) \in F_1(C)$. The PFD must have the form "$C(pf' \rightarrow pf' \circ P)$"; that is, $Z = \{ pf' \}$ and $pf = pf' \circ P$. Since $Z \subseteq Marked$ by assumption, $Root(G_c).pf'$ is marked. Since $Root(G_c).pf' \circ P$ is a descendant of $Root(G_c).pf'$, by Rule 1, $Root(G_c).pf' \circ P$ will eventually be marked. Hence $pf' \circ P \in Marked$—a contradiction with our assumption that $pf \notin Marked$.

Case 2: where $C(Z \rightarrow pf) \in F_2(C)$. By the definition of $F_2(C)$, there is a path function pf_1 in $PathFuncs(C)$ such that $C' = Ran(C, pf_1)$, $C'(W \rightarrow pf_2) \in F$, $Z = pf_1 \circ W$, and $pf = pf_1 \circ pf_2$. Let $v = Root(G_c).pf_1$. Then $l_{Cl}(v) = C'$, since $l_{Cl}(v) = Ran(C, pf_1)$ by definition. Furthermore, it follows from conditions C1 and C2 of Lemma 3 that $v.pf_2 = Root(G_c).pf_1 \circ pf_2$ and $v.pf_w = Root(G_c).pf_1 \circ pf_w$ for every $pf_w \in W$. Since $Z \subseteq Marked$ by assumption, $Root(G_c).pf_1 \circ pf_w$ $(= v.pf_w)$ is marked for every $pf_w \in W$. Thus, by Rule 2, $v.pf_2$ $(= Root(G_c).pf_1 \circ pf_2)$ will eventually be marked in view of vertex v and the PFD. Hence $pf_1 \circ pf_2 \in Marked$—again, a contradiction with our assumption that $pf \notin Marked$. □

In the above, if the C-Tree $G_c(V_c, A_c)$ is finite, then MARK(G_c, X) terminates, and it follows from Lemma 9 that Procedure 1 decides whether or not $F \models C(X \rightarrow Y)$. However, if $G_c(V_c, A_c)$ is infinite, then Procedure 1 will not always terminate. Note that $G_c(V_c, A_c)$ is infinite *iff* there are two distinct vertices v_1, v_2 on a path from $Root(G_c)$ such that $l_{Cl}(v_1) = l_{Cl}(v_2) = C'$, for some $C' \in Classes(S)$. Since $Classes(S)$ is finite, it is therefore decidable whether or not $G_c(V_c, A_c)$ is infinite.

Consequently, for the remainder of this section, we focus on the case in which $G_c(V_c, A_c)$ is infinite. The decidability of the implication problem for PFDs will be proved along the following line of argument.

> As above, let $G'_c(V'_c, A'_c)$ be the state of a C-Tree G_c after a call of the form "MARK$(G_c(V_c, A_c), X)$", where $X \subseteq PathFuncs(C)$. Given an integer c_1, we can find an integer c_2 such that, for any $pf \in PathFuncs(C)$, if (1) $len(pf) \leq c_1$ and (2) the "size" of a partial C-Tree $G(V, A)$ is c_2, then $Root(G'_c).pf$ is marked *iff* $Root(G').pf$ is marked, where $G'(V', A')$ is the state of G after a call of the form "MARK$(G(V, A), X)$". Now, since $G(V, A)$ will be finite, the call to MARK must eventually terminate. This then implies that it can be decided if $Root(G'_c).pf$ is marked without constructing the infinite C-Tree $G_c(V_c, A_c)$.

Definition 13: Let $G_1(V_1, A_1)$ and $G_2(V_2, A_2)$ be two partial C'-Trees, where $C' \in Classes(S)$. We write $G_1(V_1, A_1) \preceq G_2(V_2, A_2)$ to mean that, for every $pf \in PathFuncs(C')$, if $Root(G_1).pf$ is marked, then $Root(G_2).pf$ is marked; that is, the marked vertices in $G_1(V_1, A_1)$ are *covered* by the marked vertices in $G_2(V_2, A_2)$. If $G_1(V_1, A_1) \preceq G_2(V_2, A_2)$ and $G_2(V_2, A_2) \preceq G_1(V_1, A_1)$, then we write $G_1(V_1, A_1) \equiv G_2(V_2, A_2)$. □

Definition 14: For $C' \in Classes(S)$ and an integer l, a C'-Tree of *depth* l is a partial C'-Tree obtained from a C'-Tree by removing any vertex u and its incident arcs whenever the depth of u is greater than l. Furthermore, a C'-Tree of depth *at least* l is a partial C'-Tree that

contains a C'-Tree of depth l as a subtree with the same root. In the context of a C'-Tree $G(V, A)$ of depth at least l, we write $G(V, A)[\,l\,]$ to denote the subtree of $G(V, A)$ with the same root whose depth is l. In the absence of any such context, $G(V, A)[\,l\,]$ denotes a C'-Tree of depth l. Finally, we write $\#_V(C', l)$ to denote the number of vertices in a C'-Tree of depth l. □

For example, the tree $G(V, A)$ given in Figure 10 is an "a"-Tree of depth 3. Note that it coincides with $G_a(V_a, A_a)[3]$, for an "a"-Tree $G_a(V_a, A_a)$.

For the remainder of this section, we will also refer to the following values, as defined in the context of a class schema S, a set of PFDs F, a class $C \in \mathit{Classes}(S)$ and a finite set of path functions X, where $X \subseteq \mathit{PathFuncs}(C)$.

1. $l_1 \stackrel{def}{=} \max_{pf \in X} \mathit{len}(pf)$.
2. $l_2 \stackrel{def}{=} \max_{pf \in \{Z \cup \{pf\} \mid C'(Z \rightarrow pf) \in F\}} \mathit{len}(pf)$.
3. $L_2 \stackrel{def}{=} 1 + \Sigma_{C' \in \mathit{Classes}(S)} 2^{\#_V(C', l_2)}$. (Note that L_2 is finite since a C'-Tree of depth l_2 is finite. Also note that the value $2^{\#_V(C', l_2)}$ counts the number of different possible *true/false* assignments of $\mathit{Mark}(v_i)$ for the vertices $\{v_1, \ldots, v_n\}$ in a C'-Tree of depth l_2; that is, the number of different "marking patterns".)
4. $G_0(V_0, A_0)$ is a C-Tree of depth $l_1' + l_2 + L_2$, where l_1' is an integer such that $l_1' \geq l_1$.
5. $G_0'(V_0', A_0')$ is the state of the C-Tree G_0 following a call of the form "MARK(G_0, X)." (Note that the call to procedure MARK terminates since $G_0(V_0, A_0)$ is finite.)

The following is a key lemma, whose proof will be given in the rest of this section.

Lemma 10: $G_0'(V_0', A_0')[l_1'] \equiv G_c'(V_c', A_c')[l_1']$. □

If Lemma 10 holds, then the implication problem will be decidable by the following argument. Choose $\max_{pf \in (X \cup Y)} \mathit{len}(pf)$ as the integer l_1' (which implies $l_1' \geq l_1$ as required). Then $\mathit{Root}(G_c').pf$ is in $G_c'(V_c', A_c')[l_1']$ for every $pf \in Y$. Furthermore, $\mathit{Root}(G_c').pf$ is marked *iff* $pf \in X^+$ by Lemma 9. Hence Lemma 10 implies that $Y \subseteq X^+$ *iff* $\mathit{Root}(G_0').pf$ is marked for every $pf \in Y$. That is, the implication problem will be decidable.

Definition 15: Let $G(V, A)$ be a partial C'-Tree, where $C' \in \mathit{Classes}(S)$. A vertex $u \in V$ is *functionally complete* in $G(V, A)$ if, for every $P \in \mathit{Props}(l_{Cl}(u))$, there is an arc $u \xrightarrow{P} v \in A$ for which $l_{Cl}(v) = \mathit{Type}(l_{Cl}(u), P)$. Otherwise, u is *functionally incomplete* in $G(V, A)$. Note that if $G(V, A)$ is a C'-Tree of depth at least l, then every vertex in V whose depth is less than l is functionally complete in $G(V, A)$. □

For example, with regard to the partial "a"-Tree $G(V, A)$ in Figure 10, every vertex whose depth is less than 3 is functionally complete in $G(V, A)$. Note that $G(V, A)$ is of depth 3. As for vertices of depth 3, vertex v_9 is functionally complete, while vertices v_7, v_8, v_{10}, and v_{11} are functionally incomplete. (For example, v_7 is functionally incomplete since there is no arc of the form $v_7 \xrightarrow{\mathtt{A}} u$, even though $l_{Cl}(v_7) = \mathtt{b}$ and $\mathtt{A} \in \mathit{Props}(\mathtt{b})$.)

Let l be an integer such that $l \geq l'_1 + l_2 + L_2$, and let $G(V, A)$ be a C-Tree of depth at least l but not of depth at least $l + 1$. (Such a C-Tree is well-defined since we have assumed that the C-Tree $G_c(V_c, A_c)$ is infinite.) Let $G'(V', A')$ be the state of G after a call of the form "MARK(G, X)." Since $G'(V', A')$ is a C-Tree of depth at least l but not of depth at least $l + 1$, there is a functionally incomplete vertex $v \in V'$ whose depth is l. For each i such that $l'_1 + 1 \leq i \leq l'_1 + L_2$, there is an ancestor v' of v whose depth is i. For such an ancestor v', there corresponds a C'-Tree of depth l_2 as a subtree with root v', where $C' = l_{Cl}(v')$, since $G'(V', A')$ is a C-Tree of depth at least $l(\geq l'_1 + l_2 + L_2)$ and the depth of v' is between $l'_1 + 1$ and $l'_1 + L_2$. Let $T(v')$ be the subtree with root v'. By the choice of L_2, there are at least two distinct ancestors v_1, v_2 of v such that (1) $l_{Cl}(v_1) = l_{Cl}(v_2) = C'$ for some $C' \in \mathit{Classes}(S)$, (2) the depths of v_1 and v_2 are between $l'_1 + 1$ and $l'_1 + L_2$, and (3)

$$T(v_1)[l_2] \equiv T(v_2)[l_2]. \tag{4.1}$$

Assume without loss of generality that v_1 is an ancestor of v_2 (Figure 11 illustrates the shape of the C-Tree $G'(V', A')$ as discussed thus far), and let $G_r(V_r, A_r)$ be the tree obtained from $G'(V', A')$ by replacing the subtree $T(v_2)$ with the subtree $T(v_1)$. Then we have the following.

Lemma 11: Every $u \in V_r$ whose depth is less than l is functionally complete in $G_r(V_r, A_r)$. Also, the number of functionally incomplete vertices of depth l in $G_r(V_r, A_r)$ is less than the number of functionally incomplete vertices of depth l in $G'(V', A')$.

Proof. By definition of $G'(V', A')$, for every $u \in V'$, if the depth of u is less than l, then u is functionally complete in $G'(V', A')$. Furthermore, v_1 is a proper ancestor of v_2, since v_1 and v_2 are distinct. Thus, by replacing $T(v_2)$ with $T(v_1)$, at least the vertex $\mathit{Root}(G_r).l_{Pf}(v)$ whose depth is l becomes functionally complete in $G_r(V_r, A_r)$, and, for every $u \in V'$ whose depth is less than l, $\mathit{Root}(G_r).l_{Pf}(u)$ remains functionally complete in $G_r(V_r, A_r)$. The lemma follows. □

Lemma 12: Let $G'_r(V'_r, A'_r)$ denote the state of G_r after a call of the form "MARK(G_r, X)." Then:

$$G'_r(V'_r, A'_r) \preceq G_r(V_r, A_r) \preceq G'_c(V'_c, A'_c).$$

Proof. In the following, for a vertex $u \in V'$, the corresponding vertices $\mathit{Root}(G_r).l_{Pf}(u) \in V_r$ and $\mathit{Root}(G'_c).l_{Pf}(u) \in V'_c$ are denoted by $\langle u \rangle_r$ and $\langle u \rangle_c$, respectively, if the explicit correspondence is necessary. Also, for vertices $u \in V_r$ and $v \in V'_c$, let $T_r(u)$ and $T_c(v)$ denote the

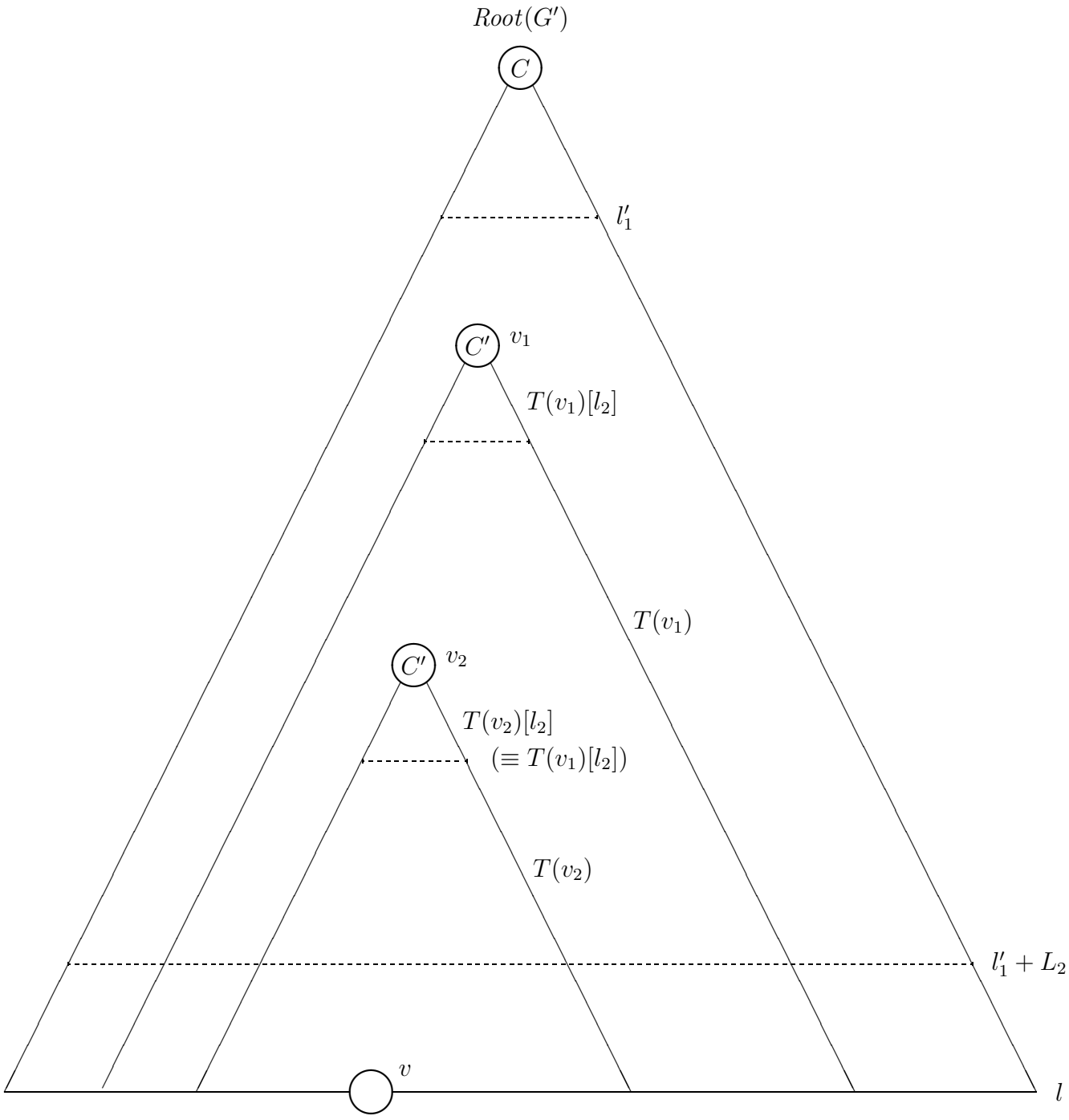

Figure 11: The C-Tree $G'(V', A')$ in the case of depth l.

subtrees of $G_r(V_r, A_r)$ and $G'_c(V'_c, A'_c)$ with roots u and v, respectively. If T is a tree and T' is a subtree of T, then $T - T'$ denotes the tree obtained by removing T' from T.

To begin, we claim the following.

Claim 1: (a) $T(v_1) \equiv T_r(\langle v_2 \rangle_r)$.

(b) $G'(V', A') - T(v_2) \equiv G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$. □

Claim 2: (a) No unmarked vertex is marked by applying Rule 1 or 2 to $G'(V', A')$.

(b) No unmarked vertex is marked by applying Rule 1 or 2 to $T(v_1)$.

(c) No unmarked vertex is marked by applying Rule 1 or 2 to $G'(V', A') - T(v_1)$. □

Claim 3: No unmarked vertex is marked by applying Rule 1 or 2 to $G'_c(V'_c, A'_c)$. □

Claim 1 follows from the construction of $G_r(V_r, A_r)$. Claim 2(a) is a simple consequence of the fact that $G'(V', A')$ is obtained by applying Rules 1 and 2 exhaustively. Claim 2(a) implies Claims 2(b) and (c) since $T(v_1)$ and $G'(V', A') - T(v_1)$ are subtrees of $G'(V', A')$, respectively. Finally, Claim 3 holds by a similar argument for Claim 2(a).

Proof that $G'_r(V'_r, A'_r) \preceq G_r(V_r, A_r)$. Note that $G'_r(V'_r, A'_r)$ is obtained by initializing $Mark(v)$ to the value *false*, for all vertices v in $G_r(V_r, A_r)$, and then executing Steps 2 and 3 of MARK. Let $G''_r(V''_r, A''_r)$ be the result of applying Steps 2 and 3 of MARK to $G_r(V_r, A_r)$ without the initialization. Since no marked vertex is changed to an unmarked status by Step 2 or 3, we have that $G'_r(V'_r, A'_r) \preceq G''_r(V''_r, A''_r)$. In the following, we prove that no unmarked vertex is changed to a marked status by applying Steps 2 and 3 of MARK to $G_r(V_r, A_r)$. This implies that $G_r(V_r, A_r) \equiv G''_r(V''_r, A''_r)$, and therefore that $G'_r(V'_r, A'_r) \preceq G_r(V_r, A_r)$.

Assume that an unmarked vertex in V_r is changed to a marked status in Step 2. Then $Root(G_r).pf$ must be unmarked for some $pf \in X$. Note that $Root(G').pf$ is marked since $G'(V', A')$ is itself the result of a call to procedure MARK. Since (1) the depth of v_2 is greater than l'_1 by definition and (2) $len(pf) \leq l_1 \leq l'_1$, the depth of $\langle v_2 \rangle_r$ is greater than the depth of $Root(G_r).pf$. Hence, $Root(G_r).pf$ is in $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$. Since $Root(G_r).pf$ is unmarked, so is $Root(G').pf$ by Claim 1(b)—a contradiction.

Now assume that an unmarked vertex in V_r is changed to a marked status in Step 3. There are two cases.

Case 1: change occurs as a consequence of Rule 1. Then there are two vertices $\langle u_1 \rangle_r, \langle u_2 \rangle_r \in V_r$ such that (1) $\langle u_1 \rangle_r$ is marked and an ancestor of $\langle u_2 \rangle_r$, and (2) $\langle u_2 \rangle_r$ is unmarked. By Claims 1(a) and 2(b), it is not the case that both $\langle u_1 \rangle_r$ and $\langle u_2 \rangle_r$ are in $T_r(\langle v_2 \rangle_r)$. By Claims 1(b) and 2(c), it is not the case that both $\langle u_1 \rangle_r$ and $\langle u_2 \rangle_r$ are in $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$. Furthermore, $\langle u_1 \rangle_r$ is an ancestor of $\langle u_2 \rangle_r$. Thus, the only possibility is that (1) $\langle u_1 \rangle_r$ is marked, an ancestor of $\langle v_2 \rangle_r$, and in $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$, and (2) $\langle u_2 \rangle_r$ is unmarked, a descendant of $\langle v_2 \rangle_r$, and in $T_r(\langle v_2 \rangle_r)$. By the former observation and Claim 1(b), u_1 is therefore marked and in $G'(V', A') - T(v_2)$. By the latter observation and Claim 1(a), there is an unmarked vertex w in $T(v_1)$. Since u_1 is marked, Rule 1 can be applied to u_1 in $G'(V', A')$. By Claim 2(a), all

descendants of u_1 should be marked in $G'(V', A')$. Since $\langle u_1 \rangle_r$ is an ancestor of $\langle v_2 \rangle_r$, u_1 is also an ancestor of v_2; that is, v_2 is a descendant of u_1. Thus v_2 is marked, and v_1 is also marked by (4.1). Then Rule 1 can be applied to v_1 in $G'(V', A')$. By Claim 2(a), all descendants of v_1 should be marked in $G'(V', A')$; that is, all vertices in $T(v_1)$ should be marked. However, this contradicts the assumption that w is unmarked and in $T(v_1)$.

Case 2: change occurs as a consequence of Rule 2. Then, for a vertex $\langle u \rangle_r \in V_r$, there is a PFD $C'(Z \rightarrow pf) \in F$ such that $C' = l_{Cl}(\langle u \rangle_r)$, $\langle u \rangle_r.pf_z$ is marked for every $pf_z \in Z$, and $\langle u \rangle_r.pf$ is unmarked. There are two more specific cases to be considered.
Case 2.1: where $\langle u \rangle_r$ is in $T_r(\langle v_2 \rangle_r)$. Since all $\langle u \rangle_r.pf$ and $\langle u \rangle_r.pf_z$ are descendants of $\langle u \rangle_r$, these vertices are in $T_r(\langle v_2 \rangle_r)$. Hence, in $T_r(\langle v_2 \rangle_r)$, the unmarked vertex $\langle u \rangle_r.pf$ can be marked by applying Rule 2 to the vertex $\langle u \rangle_r$ and the PFD. However, this contradicts Claims 1(a) and 2(b).
Case 2.2: where $\langle u \rangle_r$ is in $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$. We claim that $u.pf$ is unmarked and $u.pf_z$ is marked for every $pf_z \in Z$.

If $\langle u \rangle_r.pf$ is in $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$, then $u.pf$ is unmarked by Claim 1(b), since $\langle u \rangle_r.pf$ is unmarked. Assume that $\langle u \rangle_r.pf$ is in $T_r(\langle v_2 \rangle_r)$. Since (1) $\langle u \rangle_r$ is in $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r)$ and (2) $len(pf) \leq l_2$ by the choice of l_2, $\langle u \rangle_r.pf$ must be in $T_r(\langle v_2 \rangle_r)[l_2]$. Thus, $u.pf$ is in $T(v_2)[l_2]$. Since: (1) $\langle u \rangle_r.pf$ is unmarked, and (2) $T(v_2)[l_2] \equiv T_r(\langle v_2 \rangle_r)[l_2]$ by (4.1) and Claim 1(a), $u.pf$ is unmarked. By a similar argument, $u.pf_z$ is marked for every $pf_z \in Z$.

By the above argument, in regard to $G'(V', A')$ and the given PFD, the unmarked vertex $u.pf$ can be changed to a marked status by applying Rule 2 to vertex u, which is in contradiction with Claim 2(a). Our earlier assertion that $G'_r(V'_r, A'_r) \preceq G_r(V_r, A_r)$ then follows.

Proof that $G_r(V_r, A_r) \preceq G'_c(V'_c, A'_c)$. It follows from Lemmas 8 and 9 that

$$G'(V', A') \preceq G'_c(V'_c, A'_c). \tag{4.2}$$

This implies that $G'(V', A') - T(v_2) \preceq G'_c(V'_c, A'_c) - T_c(\langle v_2 \rangle_c)$. Thus, by Claim 1(b), $G_r(V_r, A_r) - T_r(\langle v_2 \rangle_r) \preceq G'_c(V'_c, A'_c) - T_c(\langle v_2 \rangle_c)$. What remains to prove is that $T_r(\langle v_2 \rangle_r) \preceq T_c(\langle v_2 \rangle_c)$. Since $T(v_2)[l_2] \preceq T_c(\langle v_2 \rangle_c)[l_2]$ by (4.2), it follows from (4.1) that

$$T(v_1)[l_2] \preceq T_c(\langle v_2 \rangle_c)[l_2]. \tag{4.3}$$

From this observation and Claim 1(a), it suffices to show that

$$T(v_1) - T(v_1)[l_2] \preceq T_c(\langle v_2 \rangle_c) - T_c(\langle v_2 \rangle_c)[l_2] \tag{4.4}$$

in order to prove $T_r(\langle v_2 \rangle_r) \preceq T_c(\langle v_2 \rangle_c)$. It follows from condition C1 of Lemma 3 that, for a vertex u in $T(v_1)$, there is a path function $pf \in PathFuncs(l_{Cl}(v_1))$ such that $v_1.pf = u$. To simplify the notation, assume u' denotes the corresponding vertex $\langle v_2 \rangle_c.pf$ in $T_c(\langle v_2 \rangle_c)$, and let u be a marked vertex in $T(v_1) - T(v_1)[l_2]$. Then (4.4) follows if u' must also be marked, which we prove by induction on the sequence of applications of Rules 1 and 2 during execution of Step 3 in procedure MARK which occur as a result of a call of the form "MARK(G, X)."

Basis. Initially, for each $pf \in X$, vertex $Root(G).pf$ is marked in Step 2. Since (1) the depth of v_1 is greater than l'_1, by definition, and (2) $len(pf) \leq l_1 \leq l'_1$, the depth of v_1 is greater than the depth of $Root(G).pf$. Thus, $Root(G).pf$ is in $G(V, A) - T(v_1)$; that is, there is no marked vertex in $T(v_1) - T(v_1)[l_2]$ at the end of Step 2. Hence (4.4) holds trivially.

Induction. Consider where vertex u is changed to a marked status by the ith application. By the induction hypothesis, we may assume that, for $j < i$, if the jth application of a rule in Step 3 changes vertex w in $T(v_1) - T(v_1)[l_2]$ to a marked status, then w' is also marked in $T_c(\langle v_2 \rangle_c) - T_c(\langle v_2 \rangle_c)[l_2]$. There are two cases to be considered.

Case 1: where u is changed to a marked status by Rule 1. Then there is an ancestor w of u that has already been marked. There are three subcases to be considered.
Case 1.1: where w is in $T(v_1)[l_2]$. Then w' is in $T_c(\langle v_2 \rangle_c)[l_2]$ and marked by (4.3). Since w' is an ancestor of u', vertex u' can be changed to a marked status by applying Rule 1 to w' in $G'_c(V'_c, A'_c)$. By Claim 3, u' is therefore marked.
Case 1.2: where w is in $T(v_1) - T(v_1)[l_2]$. Since w was changed to a mark status at the same time as u, w' is marked by the induction hypothesis. Hence, as in Case 1.1, u' must be marked.
Case 1.3: where w is in $G(V, A) - T(v_1)$. Since $T(v_1)$ is a tree with root v_1 and w is an ancestor of u, the assumption implies that w is an ancestor of v_1. Thus, v_1 as well as u can be changed to a marked status by applying Rule 1 to w. Furthermore, since v_1 is in $T(v_1)[l_2]$, $\langle v_2 \rangle_c$ is marked by (4.3). Since $\langle v_2 \rangle_c$ is an ancestor of u', as in Case 1.1, vertex u' must be marked.

Case 2: where u is changed to a marked status by Rule 2. Then, for an ancestor w of u, there is a PFD $C'(Z \rightarrow pf) \in F$ such that $C' = l_{Cl}(w)$, $w.pf_z$ is marked for every $pf_z \in Z$, and $w.pf = u$. Since (1) $len(pf) \leq l_2$ by choice of l_2, and (2) u is in $T(v_1) - T(v_1)[l_2]$ by assumption, the ancestor w of u must be in $T(v_1)$ in order to satisfy $w.pf = u$. Thus, each marked vertex $w.pf_z$ is also in $T(v_1)$. It can be proven along the same line of argument for Cases 1.1 and 1.2 above that each corresponding vertex $w'.pf_z$ is marked. Hence, $w'.pf$ could have been changed to a marked status by applying Rule 2 to the vertex w' and the PFD in $G'_c(V'_c, A'_c)$, and therefore, by Claim 3, $w'.pf$ is marked. (Note that $u' = w'.pf$.) This completes the induction proof; Lemma 12 now follows. □

We are now ready to prove Lemma 10. Consider the following procedure, where N is an integer such that $N \geq l'_1 + l_2 + L_2$.

Procedure 2:

Step 1. Let the initial value of $G(V, A)$ be the state $G'_0(V'_0, A'_0)$ of the C-Tree $G_0(V_0, A_0)$ of depth $l'_1 + l_2 + L_2$ which results after the call "MARK(G_0, X)." (Observe that there is no functionally incomplete vertex in V whose depth is less than $l'_1 + l_2 + L_2$.)

Step 2. for $i \leftarrow l'_1 + l_2 + L_2$ **to** N **do**

Step 3. while there is a vertex v of depth i that is functionally incomplete in $G(V, A)$ **do**

begin

Step 3.1. For such a vertex v, find two distinct ancestors v_1, v_2 of v satisfying the following three conditions (assuming, without loss of generality, that v_1 is an ancestor of v_2):

1. $l_{Cl}(v_1) = l_{Cl}(v_2)$,
2. The depths of v_1 and v_2 are between $l'_1 + 1$ and $l'_1 + L_2$, and
3. $T(v_1)[l_2] \equiv T(v_2)[l_2]$.

Step 3.2. Replace the subtree with root v_2 by the subtree with root v_1.

end □

It follows from Lemma 11 that Procedure 2 always terminates and yields a C-Tree of depth at least N for the given integer N. Let $G_N(V_N, A_N)$ the final state of $G(V, A)$ after a call to Procedure 2. Since Lemma 12 applies to each replacement in Step 3.2, we have

$$G'_N(V'_N, A'_N) \preceq G_N(V_N, A_N) \preceq G'_c(V'_c, A'_c), \tag{4.5}$$

where $G'_N(V'_N, A'_N)$ denotes the state of G_N after a call to procedure MARK of the form "MARK(G_N, X)." Now, since $\lim_{N\to\infty} G'_N(V'_N, A'_N) \equiv G'_c(V'_c, A'_c)$ by definition, (4.5) implies that

$$\lim_{N\to\infty} G_N(V_N, A_N) \equiv G'_c(V'_c, A'_c). \tag{4.6}$$

Since each replacement in Step 3.2 occurs at a deeper location than any subtree within depth l'_1 of the root, the marked status of any vertices of depth less than or equal to l'_1 remains unchanged throughout the execution of Procedure 2; that is,

$$G'_0(V'_0, A'_0)[l'_1] \equiv G_N(V_N, A_N)[l'_1]. \tag{4.7}$$

Hence, Lemma 10 follows from (4.6) and (4.7), and we have the following.

Theorem 3: Let $F \cup \{C(X \to Y)\}$ denote a set of PFDs over a given class schema S. Then it is decidable whether or not $F \models C(X \to Y)$. □

5. On Computing Closures

Let X denote a finite subset of *PathFuncs*(C) for some class schema S and class C in *Classes*(S). Although the closure X^+ may be an infinite subset of *PathFuncs*(C), the decidability proof in the previous section suggests a means of characterizing X^+. In fact, in this section, we derive an effective procedure for constructing a finite automaton which accepts X^+, and therefore prove that X^+ forms a regular set.

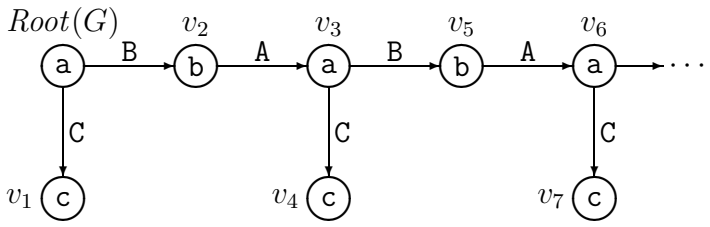

Figure 12: An "a"-Tree $G(V, A)$.

To begin, let $G_c(V_c, A_c)$ denote a C-Tree, and $G'_c(V'_c, A'_c)$ the state of G_c after a call to procedure MARK (defined in the previous section) of the form "MARK(G_c, X)." We can view G'_c as a (possibly infinite) *automaton* by letting $Root(G'_c)$ be the initial state, each marked vertex an accepting state, and each unmarked vertex a non-accepting state. Then the automaton accepts X^+ by Lemma 9 (with the simple convention that the automaton ignores any "dots" which occur in argument path functions).

If G'_c is finite, then it is clearly a finite automaton accepting X^+. For the remainder of this section, we focus on the more difficult case that arises when G'_c if infinite. As a matter of convenience, we reuse the various notation introduced by the previous chapter during the proof of Lemma 12.

Our overall strategy will be to modify G'_c to a finite automaton by redirecting various arcs. An informal example should help to clarify the main ideas behind this strategy. To begin, let S consist of the complex object types

```
a{ B: b, C: c }
b{ A: a }
c{  }
```

and consider where F consists of the single PFD

```
a( C → B.A.C )
```

and where $X = \{$ C $\}$. An "a"-Tree $G(V, A)$ (which is infinite) is illustrated in Figure 12. Now assume a call is made to procedure MARK of the form "MARK(G, X)." At the end of Step 2, vertex v_1 is marked and all other vertices are unmarked. In fact, at the end of Step 3, it is straightforward to confirm that each vertex labeled "c" (e.g., v_4 or v_7 in Figure 12) will be marked, and all other vertices unmarked. Consider each subtree of depth at least 3 ($= l_2$) with a root vertex labeled "a". Each such subtree will have the "marking pattern" illustrated in Figure 13 in which the marked vertices will correspond to the vertices with bullets in the pattern. Thus, each (infinite) subtree whose root is a vertex labeled "a" has the same marking pattern. (This will be formally proven below.) We can therefore represent the marking pattern of G by *redirecting* the destination of the out-arc of v_5 from v_6 to v_3. For this graph, we can construct a finite automaton accepting $\{$C$\}^+$ as follows: (1) let $Root(G)$ be the initial state, (2) let the two marked vertices v_1 and v_4 be accepting states, and (3) let all unmarked vertices

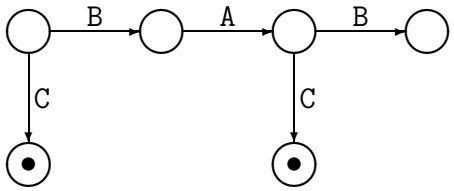

Figure 13: Marking pattern of a subtree of depth exceeding 3 with root labeled "a".

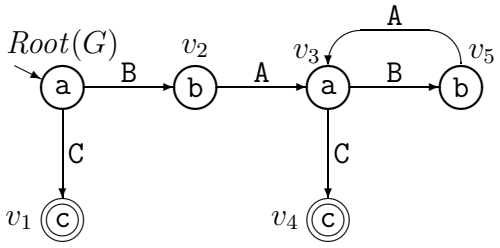

Figure 14: A finite automaton accepting $\mathtt{C}^+$.

$Root(G)$, v_2, v_3, and v_5 be non-accepting states.[7] The resulting automaton is illustrated in Figure 14. Clearly, given $pf \in PathFuncs(\mathtt{a})$, the automaton can decide in $O(len(pf))$ time whether or not $pf \in \mathtt{C}^+$.

Lemma 13: Let v_1 and v_2 be vertices in V'_c such that (1) $l_{Cl}(v_1) = l_{Cl}(v_2)$, (2) the depths of v_1 and v_2 are greater than l_1, and (3) $T_c(v_1)[l_2] \equiv T_c(v_2)[l_2]$. Then $T_c(v_1) \equiv T_c(v_2)$.

Proof. By symmetry and assumption, it suffices to show that

$$T_c(v_1) - T_c(v_1)[l_2] \preceq T_c(v_2) - T_c(v_2)[l_2]. \tag{5.1}$$

A proof of this is analogous to our prove of (4.4) in Lemma 12, and is left to Appendix 7. □

Given $pf \in PathFuncs(C)$, we need to decide whether or not $Root(G'_c).pf$ is marked; that is, if $Root(G'_c).pf$ will qualify as an accepting state in the eventual automaton. Choose l_1 as the integer l'_1, and assume $len(pf) \geq l_1 + l_2 + L_2$. Then there are two distinct ancestors v_1, v_2 of $Root(G'_c).pf$ satisfying the three conditions of Step 3.1 in Procedure 2, and it follows from Lemma 13 that $T_c(v_1) \equiv T_c(v_2)$. Thus, for every $pf' \in PathFuncs(l_{Cl}(v_1))$, $v_1.pf'$ is marked *iff* $v_2.pf'$ is marked. Now consider that there must exist path functions pf_1, pf_2 and pf_3 such that (1) $Root(G'_c).pf_1 = v_1$, (2) $Root(G'_c).pf_1 \circ pf_2 = v_2$, (3) $pf = pf_1 \circ pf_2 \circ pf_3$, and (4) $Root(G'_c).pf$ is marked *iff* $v_1.pf_3$ is marked (since $pf_3 \in PathFuncs(l_{Cl}(v_1))$). With this observation in mind, consider the following procedure for navigating within $G'_c(V'_c, A'_c)$ by following arcs labeled by properties in the sequence they occur in an argument path function.

[7] A simpler automaton exists in this case. In general, it is not part of our intention in this section to derive automatons with the fewest states.

procedure TRAVERSE($P_1.P_2.\cdots.P_n$)
Input: a path function $P_1.P_2.\cdots.P_n \in PathFuncs(C)$.
Output: a vertex $v \in V_c'$ with the same marked status as vertex $Root(G_c').pf$.

Step 1. Let $v \leftarrow Root(G_c')$.

Step 2. for $i \leftarrow 1$ **to** n **do**

begin

Step 2.1. Let $v \leftarrow v.P_i$.

Step 2.2. if there is a proper ancestor u of v such that (1) $l_{Cl}(u) = l_{Cl}(v)$, (2) the depth of u is between $l_1 + 1$ and $l_1 + L_2$, and (3) $T_1(u)[l_2] \equiv T_1(v)[l_2]$ **then** let $v \leftarrow u$.

end □

The important conditions satisfied by vertex v returned by this procedure are given by the following lemma.

Lemma 14: (a) The depth of v does not exceed $l_1 + L_2$ during TRAVERSE(pf).

(b) Assume that the three preconditions of Step 2.2 are satisfied when the value of v is v' during a call to procedure TRAVERSE. Then v' is the *shallowest* vertex on the path from $Root(G_c')$ to v' which satisfies the three preconditions. That is, for any proper ancestor v'' of v', there is no proper ancestor u'' of v'' such that (1) $l_{Cl}(u'') = l_{Cl}(v'')$, (2) the depth of u'' is between $l_1 + 1$ and $l_1 + L_2$, and (3) $T_1(u'')[l_2] \equiv T_1(v'')[l_2]$.

Proof. Part (b) of the lemma is a straightforward consequence of the fact that v is reassigned to an ancestor vertex as soon as the three preconditions of Step 2.2 are satisfied. With regard to part (a) of the lemma, assume conversely that the depth of the vertex referenced by v, say v', exceeds $l_1 + L_2$. By virtue of the value L_2, there must then exist two distinct proper ancestors v_1 and v_2 of v' satisfying the three preconditions of Step 3.1. Since v is either reassigned to a child vertex in Step 2.1 or to an ancestor vertex in Step 2.2, v must necessarily have "visited" every ancestor of v'. Thus, since v_2 is an ancestor for which the three preconditions of Step 2.2 are satisfied, procedure TRAVERSE will never visit any proper descendant of v_2, including v'—a contradiction. □

Now choose $l_1 + l_2 + L_2$ as the integer l_1', and let $G_1(V_1, A_1) = G_0'(V_0', A_0')[l_1 + l_2 + L_2]$, where $G_0'(V_0', A_0')$ is the state of a C-Tree $G_0(V_0, A_0)$ of depth $l_1' + l_2 + L_2$ following a call to procedure MARK of the form "MARK(G_0, X)." Then, by Lemma 10,

$$G_1(V_1, A_1) \equiv G_c'(V_c', A_c')[l_1 + l_2 + L_2]$$

and it follows from Lemma 14(a) that any call to another version of TRAVERSE, navigating $G_1(V_1, A_1)$, will return a vertex v which has the same marked status as $Root(G_c').pf$.

(Note that, although any vertex referenced by v in the body of the procedure is always in $G_1(V_1, A_1)[l_1 + L_2]$ by Lemma 14(a), the additional vertices in $G_1(V_1, A_1) - G_1(V_1, A_1)[l_1 + L_2]$ are still required in order to ensure that the third precondition of Step 2.2 remains effective.) Thus, since $G_0(V_0, A_0)$ is finite, $G_1(V_1, A_1)$ can be effectively constructed, and we can then use this new version of TRAVERSE as the means of deciding the marked status of any vertex in V_c'.

Let us now consider how to construct a finite automaton accepting X^+ from $G_1(V_1, A_1)$. By Lemma 14(b), we can compute the set of ordered pairs (v', u') of vertices in V_1 such that, whenever the vertex referenced by v in the body of procedure TRAVERSE becomes v' in Step 2.1, then it is changed into u' in Step 2.2. In fact, a pair (v', u') is in the set, say *Redirect*, *iff* it satisfies the following two conditions.

1. v' is the shallowest vertex on the path from $Root(G_1)$ to v' such that the three preconditions of Step 2.2 are satisfied.

2. u' is the proper ancestor of v' for which the three preconditions are satisfied.

Let (v', u') be in *Redirect*, and let w denote the parent vertex of v'. Then there is an arc $w \xrightarrow{P} v' \in A_1$ for some property P.[8] Consider when the vertex referenced by v in the body of TRAVERSE is changed from w to v' in Step 2.1 by virtue of the assignment "$v \leftarrow w.P$." By definition of the pair (v', u'), the vertex referenced by v will then subsequently be changed to u' in Step 2.2. The same effect can therefore be achieved by redirecting the destination of the arc $w \xrightarrow{P} v'$ in A_1 from v' to u' and then to perform the assignment $v \leftarrow w.P$.

By these observations, a finite automaton accepting X^+ can therefore be effectively constructed from $G_1(V_1, A_1)$ as follows.

1. Let $Root(G_1)$ be the initial state, let each marked vertex be an accepting state, and let each unmarked vertex be a non-accepting state.

2. For each pair (v', u') in *Redirect*, redirect the destination of arc $w \xrightarrow{P} v'$ from v' to u', where $w \xrightarrow{P} v' \in A_1$.

Hence, we have the following theorem and corollary.

Theorem 4: Let X denote a finite subset of $PathFuncs(C)$, where $C \in Classes(S)$ for some class schema S. Then there is an effective procedure for constructing a finite automaton that accepts X^+. □

Corollary 1: The closure X^+ is regular. □

Note that the constructed finite automaton is *essentially deterministic* in the sense that there is neither an arc labeled `Id` nor a vertex which has two or more out-arcs with the same label.[9] Thus, once the finite automaton accepting X^+ is generated, it can be decided in $O(\|Y\|)$ time whether or not $F \models C(X \rightarrow Y)$, where $\|Y\|$ is the size of Y.

[8] Since $G_1(V_1, A_1)$ is a tree, only one such arc connecting w and v' will exist.

[9] An arc labeled `Id` corresponds to an empty transition.

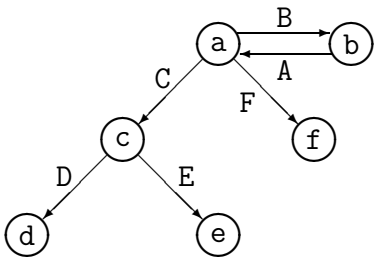

Figure 15: A SCHEMA GRAPH FOR S.

6. Polynomial Time Algorithms for Implication Problems

The decision procedure given in Section 4 is not efficient. In fact, it takes more than exponential time on the total size of S, F, and $C(X \rightarrow Y)$ in order to decide whether or not $F \models C(X \rightarrow Y)$. In this section, we will present two special cases which have polynomial time algorithms for deciding whether or not $F \models C(X \rightarrow Y)$.

To simplify matters in the remainder of this section, we first consider the problem of deciding membership of (arbitrary) path functions in $PathFuncs(C)$, for some $C \in Classes(S)$. This can be accomplished by a simple transformation of a schema graph $G_s(V_s, A_s)$ for S into a finite automaton FA_c which accepts $PathFuncs(C)$ in the sense outlined in the previous section. The transformation proceeds as follows. First, assign the vertex $v \in V_c$ such that $l_{Cl}(v) = C$ as the initial state.[10] And second, assign all vertices in V_s as accepting states. It then follows from Condition SG2 of Definition 10 that FA_c is essentially deterministic, and that, for any (not necessarily well-formed) path function pf, $v.pf \in V_s$ *iff* $pf \in PathFuncs(C)$. Hence, presuming that missing transitions will in fact go to an additional non-accepting state, FA_c decides in $O(len(pf))$ time whether or not $pf \in PathFuncs(C)$.

For example, let S consist of the following complex object types.

```
a{ B: b, C: c, F: f }
b{ A: a }
c{ D: d, E: e }
d{  }
e{  }
f{  }
```
(6.1)

Then a schema graph for S will have the form illustrated in Figure 15. The graph is transformed into the automaton $FA_{\mathtt{a}}$ by assigning the vertex labeled "a" as the initial state, and by assigning all vertices as accepting states. Then $FA_{\mathtt{a}}$ decides in $O(len(pf))$ time whether or not $pf \in PathFuncs(\mathtt{a})$. For example, it accepts the path function `B.A.C` which is in $PathFuncs(\mathtt{a})$, but rejects the path function `C.D.E.F` which is not in $PathFuncs(\mathtt{a})$ (since, as presumed, an "E" transition from the state labeled "d" goes to an non-accepting state).

Returning to the issue of efficient algorithms for the implication problem, let X be a finite

[10] Recall that v must be unique by condition SG1 of Definition 10.

subset of $PathFuncs(C)$, for some $C \in Classes(S)$, and let $G'_c(V'_c, A'_c)$ be the state of a C-Tree $G_c(V_c, A_c)$ after a call to procedure MARK of the form "MARK(G_c, X)." $G'_c(V'_c, A'_c)$ was considered as an automaton accepting $X^+ \subseteq PathFuncs(C)$ in Section 5. If $pf \in PathFuncs(C)$, then the form of Rule 1 of Step 3 in procedure MARK ensures that no non-accepting state is entered after reaching an accepting state. Hence, a simple expedient is to presume for any such automaton that, once an accepting state is entered, the remaining input is skipped and the automaton terminates with an "accept" status. We shall refer to such a machine as an *acceptor* of X^+ in the following discussion. Of course, the acceptor will only work property if the input is in $PathFuncs(C)$, but this can be resolved efficiently with the use of the automaton FA_c.

By slightly modifying procedure MARK, one can construct an acceptor of X^+.

procedure CONS(G, X)
Input: a partial C-Tree $G(V, A)$ and a finite subset X of $PathFuncs(C)$.

Step 1. Let $Root(G)$ be the initial state and let all vertices in V be non-accepting states.

Step 2. For each $pf \in X$, if $Root(G).pf$ is in V, then change $Root(G).pf$ into an accepting state, and remove all proper descendants and their incident arcs.

Step 3. Apply the following rule to $G(V, A)$ exhaustively.

> **Rule 3:** For a vertex $v \in V$, if there is a PFD $C'(Z \rightarrow pf) \in F$ such that (1) $C' = l_{Cl}(v)$, (2) each path function in Z has a form $pf_1 \circ pf_2$ such that $v.pf_1$ is an accepting state,[11] and (3) $v.pf$ is in V and is a non-accepting state, then change $v.pf$ into an accepting state and remove all proper descendants and their incident arcs. □

Since Rule 3 derives from Rules 1 and 2 in Step 3 of procedure MARK, it can be proven along the same line of argument used in the proof of Lemma 9 that the automaton produced by procedure CONS after a call of the form "CONS(G_c, X)", denoted M(G_c, X) in the remainder of this section, is an acceptor of X^+. But, since $G(V, A)$ may be infinite, a call to procedure CONS will not always terminate.

In the remainder of this section, we consider two cases in which the acceptor of X^+ can be constructed efficiently. The first case occurs if all antecedent PFDs (i.e. members of F) are key PFDs. Also, we shall continue to presume, without loss of generality, that the right-hand side of any member of F consists of a single path function.

Consider an application of Rule 3 to $G_c(V_c, A_c)$. In particular, assume there exists a vertex $v \in V$ and a PFD $C'(Z \rightarrow pf) \in F$ such that (1) $C' = l_{Cl}(v)$, (2) each path function in Z has a form $pf_1 \circ pf_2$ such that $v.pf_1$ is an accepting state, and (3) $v.pf$ is in V and is a non-accepting state. Since $C'(Z \rightarrow pf)$ is a key PFD, there exists a path function pf' such that $pf \circ pf' \in Z$. Conditions (2) and (3) then imply that there exists path functions pf_3 and pf_4 where $pf' = pf_3 \circ pf_4$ and where $v.pf \circ pf_3$ is an accepting state which is a proper descendent

[11] If $Z = \emptyset$, then condition (2) holds trivially.

of $v.pf$. Rule 3 will then update the automaton by first assigning $v.pf$ as a accepting state and then by removing all proper descendents of $v.pf$ (including $v.pf \circ pf_3$). Now let

$$\mathit{Prefix}(X) \stackrel{def}{=} \{\mathit{Root}(G_c).pf' \mid \text{ there exists } pf'' \text{ such that } pf' \circ pf'' \in X\}$$

Then $\mathit{Prefix}(X)$ is a finite subset of V_c. At the end of the second step during an invocation of procedure CONS, the initial set of accepting states will be a subset of $\{\mathit{Root}(G_c).pf \mid pf \in X\}$ (which in turn is a subset of $\mathit{Prefix}(X)$). By the observation above, the set of accepting states must continue to be a subset of $\mathit{Prefix}(X)$ during the execution of the third step. Hence, the set of accepting states in $\mathrm{M}(G_c, X)$, denoted S_{accept} in the following, is also a subset of $\mathit{Prefix}(X)$. In order to determine if a non-accepting state is to be changed to an accepting state, according to Rule 3, one need only record the set of present accepting states. That is, in order to compute S_{accept}, it suffices to keep at most $\mathit{Prefix}(X)$.

With this in mind, consider the time complexity for computing S_{accept}. Clearly, during Step 1, there is no need to construct the entire C-Tree $G_c(V_c, A_c)$. Only a subtree induced by $\mathit{Prefix}(X)$ needs to be created. Such a subtree is a partial C-Tree $G(V, A)$ with $V = \mathit{Prefix}(X)$ and $A = \{u \xrightarrow{P} v \in A_c \mid u, v \in \mathit{Prefix}(X)\}$. Since the size of $\mathit{Prefix}(X)$ is $\|X\|$, we can construct in $O(\|X\|)$ time a partial C-Tree induced by $\mathit{Prefix}(X)$. Thus, Step 1 requires $O(\|X\|)$ time. Since the partial C-Tree is essentially deterministic, Step 2 also requires $O(\|X\|)$ time. Furthermore, for a given vertex v and PFD $C'(Z \rightarrow pf)$, it can be decided in $O(\|Z \cup \{pf\}\|)$ time whether or not the PFD satisfies the three preconditions of Rule 3 with respect to v. Hence, deciding whether or not Rule 3 can be applied to a vertex requires $O(\|F\|)$ time. Since (1) the conditions of Rule 3 cannot be satisfied by any leaf vertex, and (2) the number of internal vertices in the tree is at most $\|X\| - |X| + 1$, where $|X|$ is the cardinality of X, one application of Rule 3 requires $O(\|F\|(\|X\| - |X| + 1))$ time. Also, the number of internal vertices decreases each time Rule 3 is applied to a vertex, which implies that Rule 3 is applied at most $\|X\| - |X| + 1$ times in Step 3 as a whole. Thus, Step 3 requires $O(\|F\|(\|X\| - |X| + 1)^2)$ time. Hence, S_{accept} can also be computed in that time. Note that S_{accept} is sufficient for constructing an acceptor of X^+, even if the number of non-accepting states in $\mathrm{M}(G_c, X)$ is infinite. Consequently, we have the following theorem.

Theorem 5: If F consists entirely of key PFDs, then an acceptor of X^+ can be constructed in $O(\|F\|(\|X\| - |X| + 1)^2)$ time. □

For example, let S consist of the six complex object types (6.1) above, and let F consist of the following three key PFDs.

f_1: `a( B  C.E → C )`
f_2: `a( B.A.C.E  C.D → B.A )`
f_3: `b( A → Id )`

We construct an acceptor of X^+ for the subset X of $\mathit{PathFuncs}(\mathtt{a})$ consisting of the following

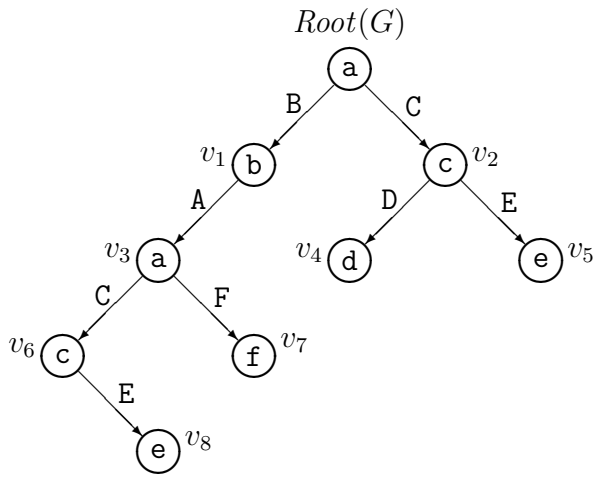

Figure 16: A TREE INDUCED BY *Prefix*(X).

path functions.

$$\begin{array}{l} \texttt{B.A.C.E} \\ \texttt{B.A.C} \\ \texttt{B.A.F} \\ \texttt{C.D} \\ \texttt{C.E} \end{array} \tag{6.2}$$

A partial "a"-Tree induced by *Prefix*(X) is illustrated in Figure 16. After the first step of procedure CONS, vertex *Root*(G) is assigned as the initial state, and all vertices as non-accepting states. In Step 2, five vertices v_4 to v_8 are changed into accepting states. For example, v_8 is changed into an accepting state by virtue of the path function $\texttt{B.A.C.E} \in X$. However, the process of changing v_6 into an accepting state (by virtue of B.A.C) will have the side-effect of removing vertex v_8.

In Step 3, the conditions of Rule 3 are satisfied by vertex *Root*(G) and PFD f_2 since (1) $l_{Cl}(\mathit{Root}(G)) = \texttt{a}$, (2) *Root*$(G)$.B.A.C $(= v_6)$ and *Root*(G).C.D $(= v_4)$ are accepting states for the left-hand side B.A.C.E, C.D of f_2, and (3) *Root*(G).B.A $(= v_3)$ is a non-accepting state for the right-hand side B.A of f_2. Applying Rule 3 to *Root*(G) and f_2 then has the effect of changing vertex v_3 into an accepting state, and of removing vertices v_6 and v_7. Rule 3 can then be applied a second time to vertex v_1 and PFD f_3. Following this application, v_1 is itself changed into an accepting state, and vertex v_3 is removed. The third and final application of Rule 3 concerns vertex *Root*(G) and PFD f_1. This will have the effect of changing vertex v_2 into an accepting state, and of removing vertices v_4 and v_5.

The resulting acceptor of X^+ appears in Figure 17. Note that *Root*(G) is the initial (non-accepting) state, and that the remaining vertices, v_1 and v_2, are accepting states.

A second case in which the acceptor of X^+ can be constructed efficiently relates to the more specific circumstance in which F consists of key PFDs which are also simple, that is, in which each key PFD in F has the single identity path function Id occurring on its right-hand

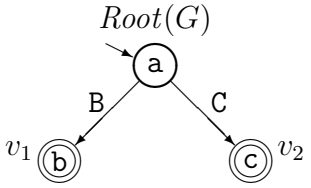

Figure 17: An acceptor of X^+.

side. In this case, S_{accept} can be computed more efficiently as explained below.

Assume we are computing S_{accept} with the use of procedure CONS in the manner outlined above, and let v be a non-accepting vertex for which no combination of a descendant of v (assuming v also qualifies as a descendent) and key PFD in F satisfies the three preconditions of Rule 3 of procedure CONS. We claim that no subsequent firing of Rule 3, during the remaining computation of S_{accept}, will assign v as an accepting state if the key PFDs in F are simple. To prove this claim, it suffices to show that no non-accepting descendant of v is changed into an accepting state by a subsequent application of Rule 3. To see this, consider any such subsequent application applied to a vertex u and a key PFD in F. Then u itself is changed into an accepting state and all its proper descendants are removed (since the right-hand side of the PFD is `Id` by assumption). Now, if u is an ancestor of v, then v will be one of the proper descendants of u which is removed, and the claim continues to hold in this case. Otherwise, if u is not an ancestor of v, then u is not a descendant of v by assumption, and, again, the claim continues to hold.

Now assume that the possible application of Rule 3 for vertices is considered is a *bottom-up* fashion. By the claim above, there will never be any need to "return" to any vertex v previously considered if, at an earlier time, it had been confirmed that no PFD in F together with v satisfied the preconditions of Rule 3. Alternatively, if the preconditions of Rule 3 had been satisfied by v and some PFD in F, then v will have been made a leaf.

These observations imply that S_{accept} can be computed by considering the possible application of Rule 3 in Step 3 of procedure CONS for each non-accepting vertex *at most once* in a bottom-up fashion. Hence, the time for Step 3 is reduced to $O(\|F\|(\|X\| - |X| + 1))$. Therefore, S_{accept} can also be computed in that time, and the following theorem holds.

Theorem 6: If every PFD in F is a key PFD which is simple, then an acceptor of X^+ can be constructed in $O(\|F\|(\|X\| - |X| + 1))$ time. □

For example, reconsider the construction of X^+ given above in which S and X consist of the six complex object types (6.1) and five path functions (6.2) respectively, but now assume F consists of the following three PFDs (note that each is a key PFD which is also simple).

f_4: `a( B.A  C.E` $\rightarrow$ `Id )`
f_5: `a( C.E  F` $\rightarrow$ `Id )`
f_6: `b( A` $\rightarrow$ `Id )`

The first two steps of procedure CONS will have the same effect on the partial "a"-Tree in Figure 16. As we have suggested for Step 3, each vertex in the tree is then considered in a bottom-up fashion to see if the vertex together with any of f_4, f_5 or f_6 satisfy the preconditions of Rule 3. To begin, assume vertex v_3 is the first vertex considered (vertex v_2 would also qualify). Clearly, v_3 and PFD f_5 satisfy the preconditions of Rule 3, and therefore v_3 is assigned as an accepting state, and vertices v_6 and v_7 are removed. Note that no further consideration of v_3 is necessary since v_3 is now a leaf vertex.

Assume vertex v_2 is the next vertex considered (vertex v_1 would now also qualify). In this case, however, neither f_4, f_5 nor f_6, together with v_2, satisfy the preconditions of Rule 3, and therefore v_2 remains as a non-accepting state. Again note that no further consideration of v_2 will be necessary.

The only possible choice for the next vertex to be considered is now v_1. In this case, v_1 and PFD f_6 satisfy the preconditions of Rule 3. v_1 is therefore assigned as an accepting state, and vertex v_3 is removed.

Finally, vertex $Root(G)$ must be considered. In this case, $Root(G)$ and PFD f_4 satisfy the preconditions of Rule 3. This will cause $Root(G)$ to be assigned as an accepting state, and all other vertices to be removed. Thus, an acceptor of X^+ consists of a single vertex $Root(G)$, which is the initial (accepting) state. Hence, X^+ coincides with $PathFuncs(\mathtt{a})$.

Note that, with the earlier construction of X^+, it would not have been possible to consider vertices in a purely bottom-up fashion. In particular, vertex v_1 and PFD f_3 would not satisfy the preconditions of Rule 3 unless the rule is applied in advance to vertex $Root(G)$ (a proper ancestor of v_1) and PFD f_2.

Note that an acceptor of X^+ constructed in a manner outlined in this section is essentially deterministic. Thus, once the acceptor X^+ is constructed, for a given PFD $C(X \rightarrow Y)$, it can be decided in $O(\|Y\|)$ time whether or not $F \models C(X \rightarrow Y)$.

7. Summary and Open Problems

In order to overcome several problems with the relational model when used for complex applications, semantic or object-oriented data models support the definition of complex object types with at least two properties. First, any object of a given type is assumed to have an identity separate from any of its parts; and second, the parts themselves may be the same or any other objects. The notion of a *path functional dependency* (or PFD) in which component attributes correspond to descriptions of property value paths in such object bases was first proposed and considered in [20]. The main contribution of this earlier work was a sound and complete axiomatization when databases may be infinite. In this paper, we have resolved a number of issues which were left open.

- We have proven that the same axiomatization remains complete when PFDs are permitted empty left-hand-sides. In our introductory comments, we reviewed an application of PFD theory which makes use of such constraints.

- We have shown that the axiomatization is not complete if logical consequence is defined with respect to finite databases only.
- We have resolved the issue of decidability of logical implication for PFDs in the affirmative. Our proof suggested that an important functional closure forms a regular set, which lead us to the derivation of an effective procedure for constructing a deterministic finite state automaton accepting the set.
- We have derived efficient polynomial time algorithms for the implication problem based on this procedure which apply in cases where antecedent PFDs are a form of complex or embedded *key* constraint.

Some issues that remain unresolved include the following.

The complexity of the general membership problem for PFDs.
Given a finite set $F \cup \{C(X \rightarrow Y)\}$ of PFDs over a class schema, is it NP-hard (or NP-complete) to decide whether or not $F \models C(X \rightarrow Y)$? The issue remains unresolved even if one restricts schema to be acyclic.

A finitely complete axiomatization.
Find a complete set of inference axioms for finite logical implication for PFDs.

Decidability and complexity issues for finite logical implication.
Given a finite set $F \cup \{C(X \rightarrow Y)\}$ of PFDs over a class schema, is it (efficiently) decidable whether or not $F \models_{\text{finite}} C(X \rightarrow Y)$? The issue remains unresolved even if $F \cup \{C(X \rightarrow Y)\}$ consists only of simple key PFDs.

In view of past experience on finite implication problems for the relational model, we expect that problems in the latter two categories will be very hard.

There is one final point worth noting about our underlying data model which relates to the concept of *generalization*. Another important feature of a semantic or object-oriented data model is that it usually allows the definition of a class (or object type) to mention at least one *superclass* (or *supertype*) — more than one if the model supports so-called *multiple inheritance*. One of the authors has extended the earlier work on PFDs in [20] to a more general model in which complex object types can also be organized in an arbitrary generalization taxonomy [19]. In particular, this later work permitted a complex object type to include an additional "`isa`" clause. For example, a `grad` complex object type for the UNIVERSITY schema could be defined as

```
grad{ Sup: prof } isa { student, prof }.
```

We wish to simply note that it is straightforward to extend the results of this paper to the more general model used in [19].

Appendix A: Proof of (5.1) in Lemma 13

By condition C1 of Lemma 3, for a vertex u in $T_c(v_1)$, there is a path function $pf \in PathFuncs(l_{Cl}(v_1))$ such that $v_1.pf = u$. For convenience, let us denote the corresponding vertex $v_2.pf$ in $T_c(v_2)$ by u'. Let u be a marked vertex in $T_c(v_1) - T_c(v_1)[l_2]$. In order to prove (5.1), it suffices to show that u' is marked, which we prove by induction on the sequence of applications of Rules 1 and 2 during execution of Step 3 in procedure MARK which occur as a result of a call of the form "MARK(G_c, X)."

Basis. Initially, for each $pf \in X$, vertex $Root(G_c).pf$ is marked in Step 2. Since (1) the depth of v_1 is greater than l_1 by assumption and (2) $len(pf) \leq l_1$, the depth of v_1 is greater than the depth of $Root(G_c).pf$. Thus $Root(G_c).pf$ is in $G_c(V_c, A_c) - T_c(v_1)$. That is, there is no marked vertex in $T_c(v_1) - T_c(v_1)[l_2]$ when Step 2 is executed. Hence (5.1) holds trivially.

Induction. Consider where vertex u is changed to a marked status by the ith application. Assume, as an induction hypothesis, that if $j < i$, then for every vertex w in $T_c(v_1) - T_c(v_1)[l_2]$ that is marked by the jth application of a rule in Step 3, w' is also marked in $T_c(v_2) - T_c(v_2)[l_2]$. There are two cases to be considered.

Case 1: where u is changed to a marked status by Rule 1. Then there is an ancestor w of u that has already been marked. There are three subcases to be considered.
Case 1.1: where w is in $T_c(v_1)[l_2]$. Then w' is in $T_c(v_2)[l_2]$, and thus marked by the assumption that $T_c(v_1)[l_2] \equiv T_c(v_2)[l_2]$. Since w' is an ancestor of u', vertex u' can be marked by applying Rule 1 to w'. By Claim 3 in the proof of Lemma 12, u' is marked in $G'_c(V'_c, A'_c)$.
Case 1.2: where w is in $T_c(v_1) - T_c(v_1)[l_2]$. Since w was marked when u is marked, w' is marked by the induction hypothesis. Hence u' is marked as in Case 1.1.
Case 1.3. Assume that w is in $G_c(V_c, A_c) - T_c(v_1)$. Since $T_c(v_1)$ is a tree with root v_1 and w is an ancestor of u, the assumption implies that w is an ancestor of v_1. Thus v_1 as well as u can be marked by applying Rule 1 to w. Furthermore, since v_1 is in $T_c(v_1)[l_2]$, v_2 is marked by the assumption that $T_c(v_1)[l_2] \equiv T_c(v_2)[l_2]$. Since v_2 is an ancestor of u', vertex u' is marked as in Case 1.1.

Case 2: where u is changed to a marked status by Rule 2. Then, for an ancestor w of u, there is a PFD $C'(Z \rightarrow pf) \in F$ such that $C' = l_{Cl}(w)$, $w.pf_z$ has been marked for every $pf_z \in Z$, and $w.pf = u$. Since (1) $len(pf) \leq l_2$ by the choice of l_2 and (2) u is in $T_c(v_1) - T_c(v_1)[l_2]$ by assumption, the ancestor w of u must be in $T_c(v_1)$ in order that $w.pf = u$. Thus, each marked vertex $w.pf_z$ is also in $T_c(v_1)$. It can be proven in the same way as Cases 1.1 and 1.2 above that each corresponding vertex $w'.pf_z$ is marked. Hence, $w'.pf$ can be marked by applying Rule 2 to w' and the PFD $C'(Z \rightarrow pf)$. By Claim 3, $w'.pf$ has already been marked in $G'_c(V'_c, A'_c)$. (5.1) then follows since $u' = w'.pf$. □

ACKNOWLEDGMENTS

The authors are grateful for the many excellent comments and suggestions by an anonymous referee.

References

[1] S. Abiteboul and S. Grumbach. COL: a logic-based langauge for complex objects. In *Proc. 1st International Conference on Extending Database Technology (EDBT)*, pages 271–293, 1988.

[2] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 159–173, June 1989.

[3] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4(4):435–454, December 1979.

[4] C. Beeri. Formal models for object-oriented databases. In *Proc. 1st International Conference on Deductive and Object-Oriented Databases*, pages 370–395, December 1989.

[5] C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5:353–382, 1990.

[6] C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In S. Even and O. Kariv, editors, *Proc. 8th ICALP*, pages 73–85. Springer-Verlag, Berlin/New York, July 1981.

[7] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. Reloop, an algebra based query language for an object-oriented database system. In *Proc. 1st Inter. Conf. in Deductive and Object-Oriented Databases*, pages 294–313, December 1989.

[8] U. Dayal. Queries and views in an object-oriented data model. In *Proc. 2nd International Workshop on Database Programming Languages*, pages 80–102, June 1989.

[9] S. K. Debray and D. S. Warren. Functional computation in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, July 1989.

[10] W. Kent. Limitations of record-based information models. *ACM Transactions on Database Systems*, 4(1):107–131, March 1979.

[11] C. Lécluse, P. Richard, and F. Velez. O_2: an object-oriented data model. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 424–433, June 1988.

[12] A. O. Mendelzon. Functional dependencies in logic programs. In *Proc. 11th International Conference on Very Large Data Bases*, pages 324–330, 1985.

[13] A. O. Mendelzon and P. T. Wood. Functional dependencies in horn clause queries. *ACM Transactions on Database Systems*, 16(1):31–55, March 1991.

[14] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A language facility for designing database-intensive applications. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[15] G. M. Shaw and S. B. Zdonik. An object-oriented query algebra. In *in Proc. 2nd Int. Workshop on Database Programming Languages*, pages 103–112, June 1989.

[16] D. W. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[17] M. F. van Bommel and G. E. Weddell. Reasoning about equations and functional dependencies on complex objects. Research Report CS-90-45, Department of Computer Science, University of Waterloo, 1990.

[18] S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 158–167, June 1991.

[19] G. E. Weddell. A theory of functional dependencies for object-oriented data models. In *Proc. 1st International Conference on Deductive and Object-Oriented Databases*, pages 150–169, December 1989.

[20] G. E. Weddell. Reasoning about functional dependencies generalized for semantic data models. *ACM Transactions on Database Systems*, 17(1):32–64, March 1992.