Annotated XML: Queries and Provenance

J. Nathan Foster Todd J. Green Val Tannen
{jnfoster,tjgreen,val}@cis.upenn.edu

Department of Computer and Information Science
University of Pennsylvania

ABSTRACT

We present a formal framework for capturing the provenance of data appearing in XQuery views of XML. Building on previous work on relations and their (positive) query languages, we decorate unordered XML with annotations from commutative semirings and show that these annotations suffice for a large positive fragment of XQuery applied to this data. In addition to tracking provenance metadata, the framework can be used to represent and process XML with repetitions, incomplete XML, and probabilistic XML, and provides a basis for enforcing access control policies in security applications.

Each of these applications builds on our semantics for XQuery, which we present in several steps: we generalize the semantics of the Nested Relational Calculus (*NRC*) to handle semiring-annotated complex values, we extend it with a recursive type and structural recursion operator for trees, and we define a semantics for XQuery on annotated XML by translation into this calculus.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Data Models

General Terms

Theory, Algorithms, Languages

Keywords

Data provenance, semirings, complex values, XML, XQuery.

1. INTRODUCTION

Recent work has shown that many of the mechanisms for evaluating queries over annotated relations—e.g., incomplete and probabilistic databases, databases with multiplicities (bags), and those carrying provenance annotations—can be unified in a general framework based on *commutative semirings* (see definition in §2). Intuitively, one of the semiring operations models alternative uses of data while the other models its joint (or dependent) use. In [16], semantics for positive relational algebra (i.e., unions of conjunctive queries) and positive Datalog were defined for relations decorated with annotations from a semiring. The same paper identified a canonical notation for provenance annotations using semiring polynomials (and formal power series) that captures, abstractly, computations in arbitrary semirings and therefore serves as a good representation for implementations [15].

This work has opened up a number of interesting avenues for investigation but its restriction to the relational model is limiting. One of the main areas that motivates work on provenance is scientific data processing. In these applications, relational data sources are often combined with data extracted from hierarchical repositories of files. XML provides a natural model for tree-structured, heterogeneous sources, but current systems for managing XML data do not provide mechanisms for decorating XML with provenance annotations and for propagating annotated data through queries. A major goal of this work is to extend the framework for semiring-annotated relations described in [16] to handle annotated XML data.

Besides provenance, our work is also motivated by applications to incomplete and probabilistic XML data. Incomplete XML has not received much attention so far (see §8), but significant work has been done on probabilistic XML. For example, in [27], the uncertainty associated with data obtained by probing the "hidden web" (i.e., data hidden behind query forms and web services) is represented using XML trees whose nodes are annotated with boolean expressions composed of independent Bernoulli event variables.

Starting from these motivations, we develop an extension of the semiring annotation framework to XML and its premier query language, XQuery [11]. Because dealing with lists and ordered XML does not seem to be related to the way we use semirings (see §8), we focus on an *unordered* variant of XML. Previous work [16] provided strong evidence that the idea of using semirings to represent annotations is robust. In this work, we describe two new results that add to this body of evidence:

- We define the semantics for a large fragment of first-order, positive XQuery—practically all of the features that do not depend on order—on semiring-annotated XML in two different ways, and show that these agree. The first approach goes by translation to an extension of the nested relational calculus [8] (*NRC*),[1] while the second uses an encoding that "shreds" XML data into a *child* relation between node identifiers, and a corresponding translation of XPath into Datalog.

- We prove a general theorem showing that the semantics of

[1] Since *NRC* is used by itself in various contexts [5, 17], this semantics is of interest even without the connection to XML.

queries commutes with the applications of semiring homomorphisms.

By instantiating our semantics using annotations formulated as polynomials over a fixed set of variables with coefficients in $\mathbb{N}$, we obtain our main contribution: a provenance framework for unordered XML data and a large class of XQuery views. We believe that this framework has practical potential: it captures an intuitive notion of provenance useful for scientific applications [15], and the size of the provenance polynomials is bounded by $O(|D|^{|q|})$ where D is the XML database and q is the XQuery program that defines the view.

Additionally, we illustrate two important applications of annotated XML: a security application that shows how to transfer *confidentiality policies* from a database to a view by organizing the clearance levels as a commutative semiring, and general *strong representation systems* for incomplete and probabilistic annotated databases that use the provenance polynomials themselves as annotations. The correctness of these systems follows from the commutation with homomorphisms theorem.

In outline, the paper is organized as follows. §2 reviews the notion of commutative semiring annotations. §3 introduces the unordered XML data model (UXML) and the corresponding fragment of XQuery (UXQuery), and describes our extension of these formalisms with semiring annotations. We defer a formal discussion of the semantics of UXQuery to §6, but illustrate its behavior on several examples. We describe applications to security and incomplete and probabilistic data in §4 and §5. The main technical results are collected in §6. There we review *NRC*, describe its extension to trees (6.1), define its semantics (6.2), give the compilation of UXQuery into this language (6.3), and state the commutation with homomorphism theorems (6.4). §7 presents an alternative definition for a fragment of UXQuery, via an encoding of UXML into relations and a translation of XPath into Datalog. §8 describes related work; we conclude with a brief discussion of ongoing and future work in §9. The long version of this abstract contains the complete definitions of each of these systems and is available as a technical report [13].

2. SEMIRING ANNOTATIONS

A *commutative semiring* $(K, +, \cdot, 0, 1)$ is an algebraic structure consisting of a set K, operations $+$ and $\cdot$, and distinguished elements $0, 1 \in K$ such that:

1. $(K, +, 0)$ and $(K, \cdot, 1)$ are commutative monoids;
2. $k_1 \cdot (k_2 + k_3) = k_1 \cdot k_2 + k_1 \cdot k_3$, and $0 \cdot k = 0$.

As shown in [16], commutative semirings and relational data fit together naturally: when each tuple in a relation is tagged with an element of K, the semantics of standard query languages can be generalized to propagate the annotations in a way that captures bag semantics, probabilistic and incomplete relations, and standard notions of provenance. An (imperfect) intuition for the meaning of these annotations is as follows: 0 means that the tuple is not present or available; $k_1 + k_2$ means that the tuple can be produced from the data described by k_1 or that described by k_2; and the annotation $k_1 \cdot k_2$ means that it requires both the data described by k_1 and that described by k_2. The annotation 1 means that exactly one copy of the tuple is available "without restrictions." In the relational setting, it was shown that the axioms of commutative semirings are forced by standard equivalences on the (positive) relational algebra [16]. In this work, we show that commutative semirings also suffice for a variety of annotated nested data and their associated query languages.

We develop our theory for arbitrary commutative semirings, but use specific semirings in various applications:

- $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$: set-based data;
- $(\mathbb{N}, +, \cdot, 0, 1)$: bag-based data;
- Positive boolean expressions: incomplete/probabilistic data (see [16] and §5);
- Confidentiality levels: see §4;
- Lineage and why-provenance (it turns out that these are different and correspond to different semirings, see [4]);
- $(\mathbb{N}[X], +, \cdot, 0, 1)$: a "universal" semiring of multivariate polynomials with coefficients in $\mathbb{N}$ and indeterminates in X.

The polynomials in $\mathbb{N}[X]$ provide a very general and informative notion of provenance[2] and, in fact, capture the generality of *all* commutative semiring calculations: any function $X \longrightarrow K$ can be uniquely extended to a semiring homomorphism $\mathbb{N}[X] \longrightarrow K$. This fact is relevant to querying since (as in [16]) by Theorem 1 and Corollary 1 below, our semantics for query answering commutes with applying homomorphisms to annotated data. This yields the principal result of our framework: a comprehensive notion of provenance for unordered XML and a corresponding fragment of XQuery.

3. ANNOTATED AND UNORDERED XML

We fix a commutative semiring K and consider XML data modified so that instead of lists of trees (sequences of elements) there are *sets* of trees. Moreover, each tree belonging to such a set is decorated with an annotation $k \in K$. Since *bags* of elements can be obtained by interpreting the annotations as multiplicities (by picking K to be $(\mathbb{N}, +, \cdot, 0, 1)$), the only difference compared to standard XML is the absence of ordering between siblings.[3] We call such data *K-annotated unordered XML*, or simply *K-UXML*. Given a domain $\mathcal{L}$ of *labels*, the usual mutually recursive definition of XML data naturally generalizes to *K-UXML*:[4]

- A *value* is either a label in $\mathcal{L}$, a tree, or a K-set of trees;
- A *tree* consists of a label together with a finite (possibly empty) K-set of trees as its "children";
- A finite *K-set of trees* is a function from trees to K such that all but finitely many trees map to 0.

In examples, we illustrate K-UXML data by adding annotations as a superscript notation on the label at the root of the (sub)tree. By convention *omitted annotations* correspond to the "neutral" element $1 \in K$.[5] Note that a tree gets an annotation only as a member of a K-set. To annotate a single tree, we place it in a singleton K-set. When the semiring of annotations is $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$ we have essentially unannotated unordered XML; we write UXML instead of $\mathbb{B}$-UXML.

In Figure 1, two K-UXML data values are displayed as trees. The source value can be written in document style as

[2] These polynomials can be used, for example, to track provenance in systems for scientific data sharing, see [15].

[3] For simplicity, we also omit attributes and model atomic values as the labels on trees having no children.

[4] In the XQuery data model, sets of labels are also values; it is straightforward to extend our formal treatment to include this.

[5] Items annotated with 0 are allowed by the definition but are useless because our semantics interprets 0 as "not present/available".

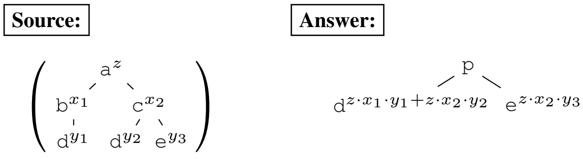


Figure 1: Simple `for` Example.

$$\begin{array}{rcl} l &\in& \mathcal{L}\\ k&\in& K\\ p &::=& l \mid \$x \mid \texttt{()} \mid \texttt{(}p\texttt{)} \mid p\texttt{,}p \mid \texttt{for}\ \$x\ \texttt{in}\ p\ \texttt{return}\ p\\ &\mid& \texttt{let}\ \$x\ \texttt{:=}\ p\ \texttt{return}\ p \mid \texttt{if (}p\texttt{=}p\texttt{) then}\ p\ \texttt{else}\ p\\ &\mid& \texttt{element}\ p\ \texttt{\{}p\texttt{\}} \mid \texttt{name(}p\texttt{)} \mid \texttt{annot}\ k\ p \mid p/s\\ s&::=& ax\texttt{::}nt\\ ax&::=& \texttt{self}\mid\texttt{child}\mid\texttt{descendant}\\ nt &::=& l\mid \texttt{*}\end{array}$$

Figure 2: K-UXQuery Syntax.

$$\begin{array}{l} \texttt{<a}^{z}\texttt{>}\ \texttt{<b}^{x_1}\texttt{>}\ \texttt{d}^{y_1}\ \texttt{</>}\\ \qquad \texttt{<c}^{x_2}\texttt{>}\ \texttt{d}^{y_2}\ \texttt{e}^{y_3}\ \texttt{</>}\ \texttt{</>} \end{array}$$

where we have abbreviated leaves `<`l`></>` as l.

We propose a query language for K-UXML called *K-UXQuery*. Its syntax, listed in Figure 2, corresponds to a core fragment of XQuery [11] with one exception: the new construct `annot` k p allows queries to modify the annotations on sets. With `annot` k p any K-UXML value can be built with the K-UXQuery constructs.

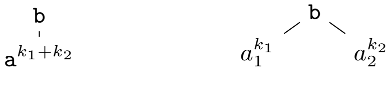
We use the following types for K-UXML and K-UXQuery:

$$t \quad ::= \quad \textit{label} \mid \textit{tree} \mid \{\textit{tree}\}$$

where *label* denotes $\mathcal{L}$, *tree* denotes the set of all trees and $\{\textit{tree}\}$ denotes the set of all finite K-sets of trees. The typing rules for selected K-UXQuery operators are given in Figure 3.

At the end of this section we discuss this syntax in more detail, and in §6.3 we present a formal semantics that uses the operations of the semiring to combine annotations. In the rest of this section, however, we illustrate the semantics informally on some simple examples to introduce the basic ideas. We start with very simple queries demonstrating how the individual operators work, and build up to a larger example corresponding to a translation of a relational algebra query.

As a first example, let $p_i = \texttt{element}\ a_i\ \texttt{\{()\}}$ for $i \in \{1,2\}$. That is, each p_i constructs a tree with no children. The query (p_1) produces the singleton K-set in which p_1 is annotated with $1 \in K$ and the query `annot` k_1 (p_1) produces the singleton K-set in which p_1 is annotated with $k_1 \cdot 1 = k_1$. We can also construct a union of K-sets: let q be `annot` k_1 (p_1)`,annot` k_2 (p_2). The result computed by q depends on whether a_1 and a_2 are the same label or different labels. If $a_1 = a_2 = \texttt{a}$, then p_1 and p_2 are the same tree and so the query then `element b {`q`}` produces the left tree below. If $a_1 \neq a_2$, then the same query produces the tree on the right.



Next, let us examine a query that uses iteration:

$$\begin{array}{l} p = \texttt{element p \{ for \$t in \$S return}\\ \qquad\qquad\qquad\quad \texttt{for \$x in (\$t)/* return}\\ \qquad\qquad\qquad\qquad \texttt{(\$x)/* \}} \end{array}$$

$$\frac{\Gamma \vdash p_1 : \{\textit{tree}\} \qquad \Gamma \vdash p_2 : \{\textit{tree}\}}{\Gamma \vdash p_1\texttt{,}p_2 : \{\textit{tree}\}}$$

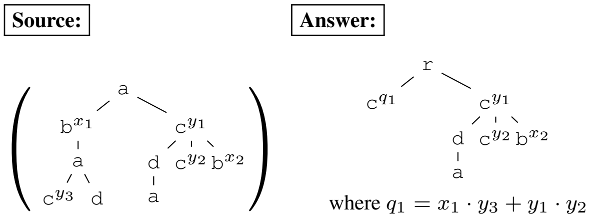
$$\frac{\Gamma \vdash p_1 : \{\textit{tree}\} \qquad \Gamma, x : \textit{tree} \vdash p_2 : \{\textit{tree}\}}{\Gamma \vdash \texttt{for}\ \$x\ \texttt{in}\ p_1\ \texttt{return}\ p_2 : \{\textit{tree}\}}$$

$$\frac{\Gamma \vdash p_1 : \textit{label} \qquad \Gamma \vdash p_2 : \textit{label} \qquad \Gamma \vdash p_3 : t \qquad \Gamma \vdash p_4 : t}{\Gamma \vdash \texttt{if (}p_1\texttt{=}p_2\texttt{) then}\ p_3\ \texttt{else}\ p_4 : t}$$

$$\frac{\Gamma \vdash p_1 : \textit{label} \qquad G \vdash p_2 : \{\textit{tree}\}}{\Gamma \vdash \texttt{element}\ p_1\ \texttt{\{}p_2\texttt{\}} : \textit{tree}} \qquad \frac{\Gamma \vdash p_1 : \textit{tree}}{\Gamma \vdash \texttt{name(}p_1\texttt{)} : \textit{label}}$$

$$\frac{\Gamma \vdash p : \{\textit{tree}\}}{\Gamma \vdash p\texttt{/}ax\texttt{::}nt : \{\textit{tree}\}} \qquad \frac{\Gamma \vdash k \in K \qquad \Gamma \vdash p : \{\textit{tree}\}}{\Gamma \vdash \texttt{annot}\ k\ p : \{\textit{tree}\}}$$

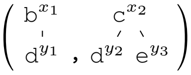
Figure 3: Selected K-UXQuery Typing Rules.



where $q_1 = x_1 \cdot y_3 + y_1 \cdot y_2$

Figure 4: XPath Example.

If `$S` is the (source) set on the left side of Figure 1, then the answer produced by p is the tree on the right in the same figure.[6] Operationally, the query works as follows. First, the outer `for`-clause iterates over the set given by `$S`. As `$S` is a singleton in our example, `$t` is bound to the tree whose root is labeled `a` and annotation in `$S` is z. Next, the inner `for`-clause iterates over the set of trees given by `($t)/*`:



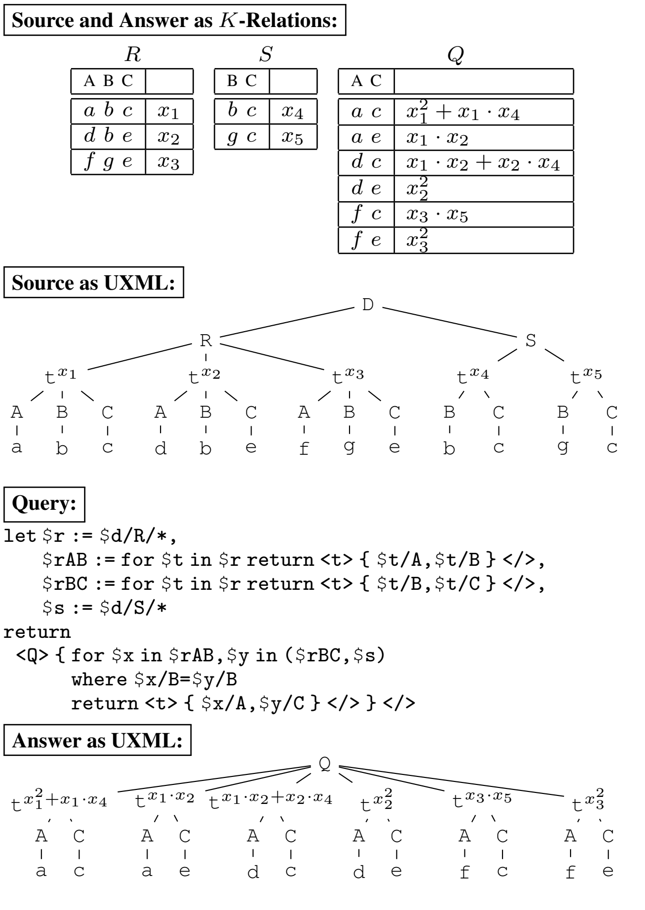
It binds `$x` to each of these trees, evaluates the `return`-clause in this extended context, and multiplies the resulting set by the annotation on `$x`. For example, when `$x` is bound to the `b` child, the `return`-clause produces the singleton set $(\texttt{d}^{y_1})$. Multiplying this set by the annotation x_1 yields $(\texttt{d}^{x_1 \cdot y_1})$. After combining all the sets returned by iterations of this inner `for`-clause, we obtain the set $(\texttt{d}^{x_1 \cdot x_1 + x_2 \cdot y_2}\texttt{,}\ \texttt{e}^{x_2 \cdot y_3})$. The final answer for p is obtained by multiplying this set by z. Note that the annotation on each child in the answer is the sum, over all paths that lead to that child in `$t`, of the product of the annotations from the root of `$t` to that child, thus recording *how* it arises from subtrees of `$S`.

Next we illustrate the semantics of XPath `descendant` navigation (shorthand `//`). Consider the query

$$r = \texttt{element r \{ \$T//c \}}$$

which picks out the set of subtrees of elements of `$T` whose label is `c`. A sample source and corresponding answer computed by r are shown in Figure 4. In §6.3 we define the semantics of the `descendant` operator using structural recursion and iteration. It

[6] Actually this query is equivalent to the shorter "grandchildren" XPath query `$S/*/*`; we use the version with a `for`-clauses to illustrate the semantics of iteration.



Source and Answer as K-Relations:

R

A B C	
$a\ b\ c$	x_1
$d\ b\ e$	x_2
$f\ g\ e$	x_3

S

B C	
$b\ c$	x_4
$g\ c$	x_5

Q

A C	
$a\ c$	$x_1^2 + x_1 \cdot x_4$
$a\ e$	$x_1 \cdot x_2$
$d\ c$	$x_1 \cdot x_2 + x_2 \cdot x_4$
$d\ e$	x_2^2
$f\ c$	$x_3 \cdot x_5$
$f\ e$	x_3^2

Source as UXML:

Query:

```
let $r := $d/R/*,
    $rAB := for $t in $r return <t> { $t/A,$t/B } </>,
    $rBC := for $t in $r return <t> { $t/B,$t/C } </>,
    $s := $d/S/*
return
 <Q> { for $x in $rAB,$y in ($rBC,$s)
       where $x/B=$y/B
       return <t> { $x/A,$y/C } </> } </>
```

Answer as UXML:

Q
$\mathtt{t}^{x_1^2+x_1\cdot x_4}$
$\mathtt{t}^{x_1\cdot x_2}$
$\mathtt{t}^{x_1\cdot x_2+x_2\cdot x_4}$
$\mathtt{t}^{x_2^2}$
$\mathtt{t}^{x_3\cdot x_5}$
$\mathtt{t}^{x_3^2}$
A C A C A C A C A C A C
a c a e d c d e f c f e

Figure 5: Relational (encoded) example.

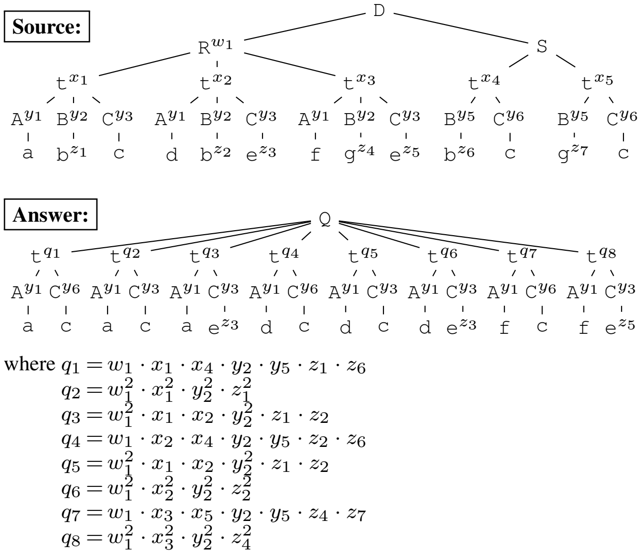
has the property that the annotation for each subtree in the answer is the sum of the products of annotations for each path from the root to an occurrence of that subtree in the source, like the answer shown here.

Now we turn to a larger example, which demonstrates how K-UXQuery behaves on an encoding of a database of relations whose tuples are annotated with elements of K (called K-relations in [16]). As a sanity check, we verify that our semantics for K-UXQuery on this data agrees agrees with the semantics given for the positive relational algebra given previously [16]. Consider the following relational algebra query

$$Q = \pi_{AC}(\pi_{AB}(R) \bowtie (\pi_{BC}(R) \cup S))$$

and suppose that we evaluate it over K-relations $R(\mathrm{A}, \mathrm{B}, \mathrm{C})$ and $S(\mathrm{B}, \mathrm{C})$ shown at the top of Figure 5. The result, cf. [16], is the K-relation $Q(\mathrm{A}, \mathrm{C})$, also shown at the top of Figure 5. For example, the annotation on $\langle d\ c\rangle$ in Q is a sum of products $x_1 \cdot x_2 + x_2 \cdot x_4$, which records that the tuple can be obtained by joining two R-tuples or, alternatively, by joining an R-tuple and an S-tuple.

The rest of Figure 5 shows the K-UXML tree that is obtained by encoding the relations R and S in an obvious way, the corresponding translation of the view definition into K-UXQuery, and the K-UXML view that is computed using K-UXQuery. Observe that the result is the encoding of the K-relation Q. The next proposition states that this equivalence holds in general. (Throughout the paper we abuse notation and conflate the syntax and semantics of expressions—i.e., we write e instead of $[\![e]\!]$.)



Source:

D
$\mathtt{R}^{w_1}$
S
$\mathtt{t}^{x_1}$
$\mathtt{t}^{x_2}$
$\mathtt{t}^{x_3}$
$\mathtt{t}^{x_4}$
$\mathtt{t}^{x_5}$
$\mathtt{A}^{y_1}$ $\mathtt{B}^{y_2}$ $\mathtt{C}^{y_3}$ $\mathtt{A}^{y_1}$ $\mathtt{B}^{y_2}$ $\mathtt{C}^{y_3}$ $\mathtt{A}^{y_1}$ $\mathtt{B}^{y_2}$ $\mathtt{C}^{y_3}$ $\mathtt{B}^{y_5}$ $\mathtt{C}^{y_6}$ $\mathtt{B}^{y_5}$ $\mathtt{C}^{y_6}$
a $\mathtt{b}^{z_1}$ c d $\mathtt{b}^{z_2}$ $\mathtt{e}^{z_3}$ f $\mathtt{g}^{z_4}$ $\mathtt{e}^{z_5}$ $\mathtt{b}^{z_6}$ c $\mathtt{g}^{z_7}$ c

Answer:

Q
$\mathtt{t}^{q_1}$
$\mathtt{t}^{q_2}$
$\mathtt{t}^{q_3}$
$\mathtt{t}^{q_4}$
$\mathtt{t}^{q_5}$
$\mathtt{t}^{q_6}$
$\mathtt{t}^{q_7}$
$\mathtt{t}^{q_8}$
$\mathtt{A}^{y_1}$ $\mathtt{C}^{y_6}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_3}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_3}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_6}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_3}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_3}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_6}$ $\mathtt{A}^{y_1}$ $\mathtt{C}^{y_3}$
a c a c a $\mathtt{e}^{z_3}$ d c d c d $\mathtt{e}^{z_3}$ f c f $\mathtt{e}^{z_5}$

where $q_1 = w_1 \cdot x_1 \cdot x_4 \cdot y_2 \cdot y_5 \cdot z_1 \cdot z_6$
$q_2 = w_1^2 \cdot x_1^2 \cdot y_2^2 \cdot z_1^2$
$q_3 = w_1^2 \cdot x_1 \cdot x_2 \cdot y_2^2 \cdot z_1 \cdot z_2$
$q_4 = w_1 \cdot x_2 \cdot x_4 \cdot y_2 \cdot y_5 \cdot z_2 \cdot z_6$
$q_5 = w_1^2 \cdot x_1 \cdot x_2 \cdot y_2^2 \cdot z_1 \cdot z_2$
$q_6 = w_1^2 \cdot x_2^2 \cdot y_2^2 \cdot z_2^2$
$q_7 = w_1 \cdot x_3 \cdot x_5 \cdot y_2 \cdot y_5 \cdot z_4 \cdot z_7$
$q_8 = w_1^2 \cdot x_3^2 \cdot y_2^2 \cdot z_4^2$

Figure 6: Extended Annotations Example.

PROPOSITION 1. *Let Q be be a query in positive relational algebra, and I a K-relational database instance. Let v be the K-UXML encoding of I, and p be the translation of Q into K-UXQuery. Then $p(v)$, computed according to K-UXQuery, encodes $Q(I)$, the K-relation computed according to the semantics in [16].*

In a K-relation, annotations *only* appear on tuples. In our model for annotated UXML data, however, every internal node carries an annotation (recall that, according to our convention, every node in Figure 5 depicted with no annotation carries the "neutral" element $1 \in K$). Therefore, we have more flexibility in how we annotate source values—besides tuples, we can place annotations on the values in individual fields, on attributes on the relations themselves, and even on the whole database! It is interesting to see how, even for a query that is essentially relational, these extra annotations participate in the calculations. We have worked this out in the final example of this section, see Figure 6. The query is the same as in Figure 5 but the source data has additional annotations. Note how the expressions annotating the tuple nodes in the answer involve many non-tuple annotations from the source.

So far we have assumed that the annotations belong to an arbitrary commutative semiring K and we looked at the expressions that equate $q_1, \ldots, q_8$ in Figure 6 as calculations in K. However, if we work with the semiring of polynomials $K = (\mathbb{N}[X], +, \cdot, 0, 1)$ where we think of the source annotations as indeterminates ("provenance tokens") and take

$$X := \{w_1, x_1, \ldots, x_5, y_1, \ldots, y_6, z_1, \ldots, z_7\}$$

then the expressions that equate $q_1, \ldots, q_8$ are the *provenance polynomials* that annotate the tuple nodes in the answer. This kind of provenance shows, for example, that some of the tuples in the answer use source data annotated with z_1 or y_5 although these do not appear explicitly in the annotations of the answer attributes or values in the tuples. The annotations in a particular semiring K can then be computed by evaluating these polynomials in K. Corollary 1 (commutation with homomorphisms) guarantees that the result will be the same as that obtained via the semantics on K-UXML values.

Note also that we can obtain the answer shown in Figure 5 simply by setting all the indeterminates except for $x_1, \ldots, x_5$ to 1 and then simplifying using the semiring laws. When we set these indeterminates to 1, some subtrees which were distinguished by annotations become now identified (q_1 and q_2, q_4 and q_5); this explains the sums in the annotations of the answer in Figure 5.

The semantics in §6 allows us to prove the following upper bound:

PROPOSITION 2. *If v is a UXML value annotated with indeterminates from a set X and p is a UXQuery, then computing $p(v)$ according to the $\mathbb{N}[X]$-UXQuery semantics produces an $\mathbb{N}[X]$-UXML value such that the size of any of the provenance polynomials that annotate $p(v)$ is $O(|v|^{|p|})$.*

K-UXQuery vs. XQuery Although UXQuery only contains core operators, more complicated syntactic features such as `where`-clauses that we used in the examples above can be *normalized* into core queries using standard translations [11]. For example, the `where`-clause `where $x/B=$y/B` from Figure 5 normalizes to:

```
for $a in $x/B/* return for $b in $y/B/* return
if (name($a)=name($b)) then ... else ()
```

Our language includes only the downward XPath axes, since the other axes can be compiled into this fragment [24]. To simplify our formal system, we also do not identify a value with the singleton set containing it. This is inessential but it simplifies the compilation in §6.3. In examples we often elide the extra set constructor when it is clear from context—e.g., we wrote `$x/A` above, not `($x)/A`.

Unlike these minor differences, we made two essential restrictions in the design of K-UXQuery. The first has to do with order—we omit `orderby` and other operators whose semantics depends on position, since these do not make sense on unordered data. The second essential restriction is to *positive* queries—e.g., the conditional expression only tests the equality of labels; see §6.1 for further discussion.

4. A SECURITY APPLICATION

We can model *confidentiality policies* using commutative semirings. For example, the total order $\mathcal{C}\ :\ \mathsf{P} < \mathsf{C} < \mathsf{S} < \mathsf{T} < 0$ describes the following levels of "clearance": P = public, C = confidential, S = secret, and T = top-secret. It is easy to see that $(\mathcal{C}, \min, \max, 0, \mathsf{P})$ is a commutative semiring.[7] We add 0 as a separate element. It is needed because items in a K-UXML set with annotation 0 are interpreted as *not* belonging to the set (i.e., 0 is so secret, it isn't even there!), and we do not want to lose data tagged as T completely.

Our framework solves the following problem. Suppose that an XML database has been manually annotated with security information specifying what clearance one must have for each data subtree they wish to see. Now we use XQuery to produce views of this database. We would like to compute automatically clearance annotations for the data in a view, based on *how* that data was obtained from the already annotated data in the original database. The two operations of the clearance semiring correspond to *alternatives* in obtaining the view data, in which case the minimum clearance among them suffices, and to *joint necessities*, in which case the maximum clearance among them is needed.

We give an example that shows that our annotated XML model is a particularly flexible framework for such clearance specifications. Consider the source data in Figure 6, which in fact encodes a relational database but where we have much more annotation flexibility than in the [16] model where only tuples are annotated. We annotate with elements from $\mathcal{C}$ as follows $w_1 := \mathsf{C}$ (the entire relation `R` is confidential), $x_2 := \mathsf{S}$ (in addition, this tuple is secret), and $y_5 := \mathsf{T}$ (all values of attribute `B` in relation `S` are top-secret), and finally, the rest of the annotations are P (which plays the role of 1 in this semiring).

Q

A	C		
a	c	$w_1 \cdot y_5 + w_1^2 = \mathsf{C} \cdot \mathsf{T} + \mathsf{C}^2 =$	C
a	e	$w_1^2 \cdot x_2 = \mathsf{C}^2 \cdot \mathsf{S} =$	S
d	c	$w_1 \cdot x_2 \cdot y_5 + w_1^2 \cdot x_2 = \mathsf{C} \cdot \mathsf{S} \cdot \mathsf{T} + \mathsf{C}^2 \cdot \mathsf{S} =$	S
d	e	$w_1^2 \cdot x_2^2 = \mathsf{C}^2 \cdot \mathsf{S} =$	S
f	c	$w_1 \cdot y_5 = \mathsf{C} \cdot \mathsf{T} =$	T
f	e	$w_1^2 = \mathsf{C}^2 =$	C

Figure 7: Security Clearance Example.

The result of the view/query in Figure 5 when applied to this data is the $\mathcal{C}$-UXML encoding of a relation in which only the annotations on the tuples are different from $1 = \mathsf{P}$ (this is because the query projects out the attribute `B`, otherwise we could have had non-P annotations inside the tuples). We show this answer as an annotated relation in Figure 7. We also show there the polynomials that would annotate the tuples if we would do the calculations in the provenance semiring $\mathbb{N}[w_1, x_2, y_5]$. These help understand *how* the resulting clearances are computed since it is a consequence of Corollary 1 (commutation with homomorphisms) that by evaluating the provenance polynomials in $\mathcal{C}$ under the valuation $w_1 := \mathsf{C}, x_2 := \mathsf{S}, y_5 := \mathsf{T}$ we get the same result as the $\mathcal{C}$-UXQuery semantics.

Going back to the security application, for the data in the view, confidential clearance gives access to the first and last tuple, secret clearance to all but the fifth tuple, etc. Note how the top-secret annotation of the attribute `B` in `S` affects just three of the tuples in the answer and how in two of those cases the tuples are still available to lower clearances because they can be also produced with data from `R` only.

In the example above the semiring of clearances is a total order but this can be generalized to non-total orderings, provided they form a *distributive lattice*. The distributivity ensures that views that we consider equivalent actually compute the same clearance for the results. This follows from the following proposition which generalizes a similar result in [16] for relations and positive relational algebra.

PROPOSITION 3. *If two UXQueries are equivalent on all UXML inputs and K is a distributive lattice then the queries are equivalent on all K-annotated UXML inputs.*

5. INCOMPLETE AND PROBABILISTIC K-UXML

Commutative semirings can also be used to model incomplete and probabilistic databases for unordered XML data, even with repetitions. An incomplete UXML database is a set of *possible worlds*, each of which is itself a UXML (i.e. a $\mathbb{B}$-UXML) database. For repetitions (multiplicities) the possible worlds are $\mathbb{N}$-UXML databases. More generally, we treat here incomplete K-UXML databases for arbitrary commutative semirings K. It turns out that by using provenance annotations we can construct a powerful system for representing and querying incomplete K-UXML databases.

Recall that provenance polynomials are elements of the commutative semiring $(\mathbb{N}[X], +, \cdot, 0, 1)$—i.e. polynomial expressions over variables X with natural number coefficients [16]. For any commutative semiring K, provenance polynomials are "universal"

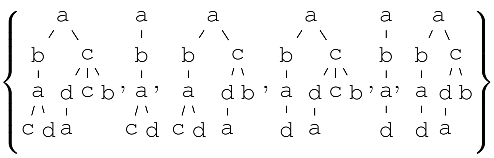
[7] Note that the natural order [16] on this semiring is actually the opposite of the clearance order.

in the sense that any function $f : X \to K$ (we call f a *valuation*) extends uniquely to a semiring homomorphism $f^\star : \mathbb{N}[X] \to K$. We exploit this to construct a representation system for incomplete K-UXML data. We first fix a semiring K, and a set of variables X. We call a v in $\mathbb{N}[X]$-UXML a *representation*. Next we define a function Mod_K that maps a representation v in $\mathbb{N}[X]$-UXML to the *set* of K-UXML instances that can be obtained by applying K-valuations to the variables in X—i.e., $\mathrm{Mod}_K(v)$ is $\{f^\star(v) : f : X \to K\}$, the set of *possible worlds* v represents.

As an example, let v be the source tree in Figure 4. To streamline the example, we will set the x_1 and x_2 annotations to 1, leaving just the annotations y_1, y_2, y_3 on the subtrees labeled c.

For $K = \mathbb{B}$, the set of possible worlds represented by v is the following set of UXML values:

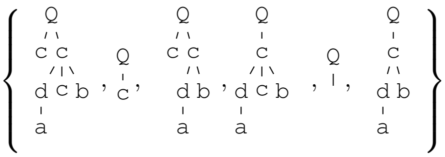
$\mathrm{Mod}_{\mathbb{B}}(v) =$



Each tree in $\mathrm{Mod}_{\mathbb{B}}(v)$ is obtained using a valuation from the y_is to $\mathbb{B}$—e.g., for the rightmost tree in this display, the valuation maps y_1 to true and y_2 and y_3 to false.

Now consider querying such an incomplete UXML database. In general, given an XQuery p, we would like the answer to be (semantically) the set of all K-UXML instances obtained by evaluating p over each K-UXML instance in the set of possible worlds represented by v—i.e., $p(\mathrm{Mod}_K(v))$ is $\{p(v') : v' \in \mathrm{Mod}_K(v)\}$. Returning to the representation v above and using p, the query in Figure 4, we have:

$p(\mathrm{Mod}_{\mathbb{B}}(v)) =$

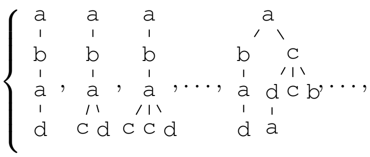


As usual in incomplete databases, we do not wish to return this set, which may be large in general. Instead, we would like a *representation* of it. By Corollary 1 below, it turns out that such a representation is obtained by evaluating p over v with $\mathbb{N}[X]$-UXQuery semantics. In general, we have that $p(\mathrm{Mod}_K(v)) = \mathrm{Mod}_K(p(v))$. Indeed, the specific answer for this example shown in Figure 4 is the representation of $p(\mathrm{Mod}_{\mathbb{B}}(v))$. Using the terminology of incomplete databases, we say that $\mathbb{N}[X]$-UXML is a *strong representation system* [19, 1] for K-UXQuery and K-UXML data.

For simpler K, the full power of $\mathbb{N}[X]$ may not be needed. For example, when $K = \mathbb{B}$, we can use annotations from the semiring $(\mathsf{PosBool}(B), \vee, \wedge, \mathsf{false}, \mathsf{true})$ of positive Boolean expressions over a set B of variables (i.e., the expressions involve only B, disjunction, conjunction, and constants for true and false).[8] This corresponds to an XML analogue of the Boolean c-tables [19] used in incomplete databases. Valuations $\nu : B \to \mathbb{B}$ extend uniquely to homomorphisms $\nu^\star : \mathsf{PosBool}(B) \to \mathbb{B}$, so the definition above of $\mathrm{Mod}_{\mathbb{B}}$ still makes sense. Indeed, it follows (again from the commutation with homomorphisms in § 6.4) that $\mathsf{PosBool}(B)$-UXML is a *strong representation system* for UXQuery and $\mathbb{B}$-UXML (i.e., ordinary UXML) and that we can transform an $\mathbb{N}[B]$-UXML representation into $\mathsf{PosBool}(B)$-UXML representation by applying the obvious homomorphism. It can be shown that PosBool works not just for $\mathbb{B}$ but for incomplete L-UXML for any distributive lattice L, in particular the ones used for the security application in §4.

Another instance of our general result is that $\mathbb{N}[X]$-UXML also provides a strong representation system for UXML with repetitions. For example, if we let v be the same tree as above, and pick $K = \mathbb{N}$, then the set of possible worlds is the following:

$\mathrm{Mod}_{\mathbb{N}}(v) =$



Note that children may be repeated—e.g., the third tree in this display has a subtree with two children c; this is obtained from a valuation that maps y_2 to 2.

Probabilistic data can also be modeled using semiring annotations. Again we use as representations $\mathbb{N}[X]$-UXML values and *all* the worlds corresponding to valuations $f : X \to K$. But now we consider such a valuation as the conjunction of independent events, $\{f(x) = k\}$ one for each x. The probability of each independent event can be computed from some probability distribution on K. For example, if $K = \mathbb{B}$ we can use Bernoulli distributions, if $K = \mathbb{N}$ we can use $\mathbf{Pr}[f(x) = n] = 1/2^n$ for $n > 0$, and 0 for $f(x) = 0$, etc. It follows again that we have a *strong representation system* this time for probability distributions on all the possible instances. For $K = \mathbb{B}$, more generally for distributive lattices, it suffices again to use PosBool expressions. Since tree pattern queries are expressible in UXQuery, we get the query evaluation algorithm in [27] as a particular case.

6. SEMANTICS VIA COMPLEX VALUES

In this section we develop our formal semantics for K-UXQuery by translation into a data model and query language for complex values. Trees can be understood as data values built recursively using pairing and collection constructions (see e.g., [6, 26]). For UXML trees, the collections are sets. This suggests defining trees as *complex values*, as data values built using pairing and sets, nested arbitrarily.

We develop our semantics in several steps. First, we generalize the semantics of *NRC* to handle semiring-annotated values. We then extend the calculus with a recursive tree type and structural recursion operator on trees. This operator is needed to express the `descendant` operator of K-UXQuery.[9] Finally, we use this calculus as a compilation target for K-UXQuery. At the end of the section, we prove a correctness theorem, stating that the semantics commutes with semiring homomorphisms, and explore some of its broader implications.

6.1 Complex Values and Trees

We start from the (positive) Nested Relational Calculus [8]. The types of *NRC* are:

$$t \quad ::= \quad \mathit{label} \mid t \times t \mid \{t\}$$

Complex values are built with the following constructors:

$$v \quad ::= \quad l \mid (v, v) \mid \{v\} \mid v \cup v \mid \{\}$$

We abbreviate $\{v_1\} \cup \cdots \cup \{v_n\}$ as $\{v_1, \ldots, v_n\}$—e.g., $(l_1, \{l_2, l_3\})$ is a complex value of type $\mathit{label} \times \{\mathit{label}\}$.

[8] We also identify those expressions which yield the same truth value for all Boolean assignments of the variables in B (to permit simplifications).

[9] When the nesting depth of the XML documents is bounded, the structural recursion operator (and the recursive tree type) are not needed, see [10].

The restriction to the *positive* fragment of the calculus is embodied in the typing rule for conditionals—we only compare label values. It is shown in [8] that equality tests for arbitrary sets can be used to define non-monotonic operations (i.e., difference, intersection, membership, and nesting). This restriction is essential for the semantics of *NRC* on annotated complex values because semirings do not contain features for representing negation.

The crucial *NRC* operation is the big-union operator: $\bigcup(x \in e_1)\ e_2$. It computes the union of the family of sets defined by e_2 indexed by x, where x takes each value in the set e_1. For example, the first relational projection is expressed as follows

$$\mathsf{project}_1\ R \triangleq \bigcup(x \in R)\ \{\pi_1(x)\}.$$

To represent trees, we extend the calculus with a constructor $\mathsf{Tree}(a, C)$ where a is the label and C the set of immediate subtrees. Trees of the form $\mathsf{Tree}(a, \{\})$ are leaves. The typing rule for the tree constructor is given by:

$$\frac{\Gamma \vdash v_1 : \mathit{label} \qquad \Gamma \vdash v_2 : \{\mathit{tree}\}}{\Gamma \vdash \mathsf{Tree}(v_1, v_2) : \mathit{tree}}$$

where *tree* is a new type. It is easy to see that the values of type *tree* and $\mathit{label} \times \{\mathit{tree}\}$ are in a 1-1 correspondence. In one direction this isomorphism is witnessed by $\mathsf{Tree}(\pi_1(P), \pi_2(P))$, where P is a pair. To express the other direction, we extend the calculus with two new operations, $\mathit{tag}(-)$ and $\mathit{kids}(-)$ that return the root tag and the set of subtree children of the root, respectively. The mapping from trees to pairs is then given by $(\mathit{tag}(T), \mathit{kids}(T))$, where T is a tree. Hence, *semantically*, the *tree* type is *recursive*.[10] In the spirit of [26] we add an operation for *structural recursion* on trees:

$$\frac{\Gamma, x : \mathit{label}, y : \{t\} \vdash e_1 : t \qquad \Gamma \vdash e_2 : \mathit{tree}}{\Gamma \vdash (\mathit{srt}(x, y).\, e_1)\ e_2 : t}$$

Its semantics obeys the equation

$$\begin{aligned}(\mathit{srt}(x, y).\, e_1)\ \mathsf{Tree}(e_2, e_3)\ &= \\ e_1[x := e_2, y := \textstyle\bigcup(z \in e_3)\ (\mathit{srt}(x, y).\, e_1)\ z]&\end{aligned} \qquad (1)$$

where the notation $e[x := e']$ denotes substitution of e' for x in e. For example, the query

$$(\mathit{srt}(x, y).\, \{x\} \cup \mathsf{flatten}\ y)\ t$$

where $\mathsf{flatten}\ W \triangleq \bigcup(w \in W)\ w$ returns the set of atoms in t. We denote this query language by *NRC* + *srt*.

6.2 Semantics for *NRC* + *srt*

Next we show how to decorate complex values (and trees) with semiring annotations, and generalize *NRC*+*srt* to operate on annotated values. Again we fix a commutative semiring $(K, +, \cdot, 0, 1)$. Dealing with complex values annotated with elements from K requires a different semantics for the type $\{t\}$. The usual semantics is the set of finite subsets of $[\![t]\!]$. Instead, the semantics of $[\![\{t\}]\!]_K$ is defined as the set of functions $f : [\![t]\!]_K \to K$ with *finite support*, i.e., such that $\mathsf{supp}(f) := \{a \in [\![t]\!]_K \mid f(a) \neq 0\}$ is finite. We call elements of $[\![\{t\}]\!]_K$ *K-collections*. With $K = \mathbb{B}$ we obtain the usual semantics as finite subsets; with $K = \mathbb{N}$ we get bags.

K-complex values are obtained by arbitrarily nesting pairing and K-collections. We define new semantics for the *NRC* constructors: the singleton constructor $[\![\{v\}]\!]_K$ is the function that maps $[\![v]\!]_K$ to 1 and everything else to 0; $[\![\{\}]\!]_K$ is the constant function that maps everything to 0; and $[\![v_1 \cup v_2]\!]_K$ is the pointwise K-addition of $[\![v_1]\!]_K$ and $[\![v_2]\!]_K$. In order to express all K-collections in the calculus, we extend *NRC* with an operation for multiplying the annotations on the elements of K-collections by the "scalar" k in K. It is written $k\, e$ and has the following typing rule:

$$[\![l]\!]^\rho_K = l \qquad [\![x]\!]^\rho_K = \rho(x) \qquad [\![\{\}]\!]^\rho_K(x) = 0_K$$

$$[\![\{e\}]\!]^\rho_K(x) = \mathit{if}\ x = [\![e]\!]^\rho_K\ \mathit{then}\ 1_K\ \mathit{else}\ 0_K$$

$$[\![e_1 \cup e_2]\!]^\rho_K(x) = [\![e_1]\!]^\rho_K(x) + [\![e_2]\!]^\rho_K(x)$$

$$\frac{[\![e_1]\!]^\rho_K = s_1}{[\![\bigcup(x \in e_1)\ e_2]\!]^\rho_K(y) = \sum_{v \in \mathsf{dom}(s_1)} s_1(v) \cdot [\![e_2]\!]^{\rho[x \leftarrow v]}_K(y)}$$

$$\frac{\mathit{if}\ [\![e_1]\!]^\rho_K = [\![e_2]\!]^\rho_K\ \mathit{then}\ [\![e_3]\!]^\rho_K\ \mathit{else}\ [\![e_4]\!]^\rho_K = v}{[\![\mathit{if}\ e_1 = e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4]\!]^\rho_K = v}$$

$$\frac{[\![e_1]\!]^\rho_K = v_1 \qquad [\![e_2]\!]^\rho_K = v_2}{[\![(e_1, e_2)]\!]^\rho_K = (v_1, v_2)} \qquad \frac{[\![e]\!]^\rho_K = (v_1, v_2) \qquad i \in \{1, 2\}}{[\![\pi_i(e)]\!]^\rho_K = v_i}$$

$$\frac{[\![e_1]\!]^\rho_K = l \qquad [\![e_2]\!]^\rho_K = s}{[\![\mathsf{Tree}(e_1, e_2)]\!]^\rho_K = \mathsf{Tree}(l, s)}$$

$$\frac{[\![e]\!]^\rho_K = \mathsf{Tree}(l, s)}{[\![\mathit{kids}(e)]\!]^\rho_K = s} \qquad \frac{[\![e]\!]^\rho_K = \mathsf{Tree}(l, s)}{[\![\mathit{tag}(e)]\!]^\rho_K = l}$$

$$\frac{[\![e_2]\!]^\rho_K = \mathsf{Tree}(l, s) \qquad [\![\bigcup(z \in s)\ (\mathit{srt}(x, y).\, e_1)\ z]\!]^\rho_K = s'}{[\![(\mathit{srt}(x, y).\, e_1)\ e_2]\!]^\rho_K = [\![e_1]\!]^{\rho[x := l, y := s']}_K}$$

Figure 8: Semantic Equations for *NRC*$_K$ + *srt*

$$\frac{\Gamma \vdash k \in K \qquad \Gamma \vdash e : \{t\}}{\Gamma \vdash k\, e : \{t\}}$$

We call the calculus extended with this operator *NRC*$_K$. The set of K-complex values are constructed using:

$$v \quad ::= \quad l \mid (v, v) \mid k\,\{v\} \mid v \cup v \mid \{\}$$

and, as above, we abbreviate K-collections using the following notation: $\{v_1^{k_1}, \ldots, v_n^{k_n}\} \triangleq k_1\,\{v_1\} \cup \cdots \cup k_n\,\{v_n\}$. Determining the right semantics for the $\bigcup(x \in e_1)\ e_2$ operation is more challenging. In Appendix A we explain this semantics in the context of a general theory of collection types [8, 21]. Here we give the semantics semi-formally.

Let e_1 have type $\{t_1\}$ and e_2 have type $\{t_2\}$ (whenever x has type t_1). Let $X = [\![t_1]\!]_K$ and $Y = [\![t_2]\!]_K$. Then $[\![e_1]\!]_K$ is a function $f : X \to K$ with finite support $\mathsf{supp}(f) = \{x_1, \ldots, x_n\}$. In general e_2 depends on x so for each x_i we have a corresponding semantics for e_2, i.e., a function $g_i : Y \to K$. Using this function we define for each $y \in Y$

$$[\![\bigcup(x \in e_1)\ e_2]\!]_K(y) \ \triangleq\ \sum_{i=1}^{n} f(x_i) \cdot g_i(y)$$

Since each g_i has finite support, so does $[\![\bigcup(x \in e_1)\ e_2]\!]_K$.

The semantics of the other operations inherited from positive *NRC* is straightforward (it is essential that the equality test does not involve K-collections and therefore additional annotations). For

[10] $\mathsf{Tree}(-, -)$, $\mathit{tag}(-)$ and $\mathit{kids}(-)$ are an instance of a standard technique for handling recursive types in functional languages.

example,

$$\begin{aligned}\textsf{flatten}\ \{\{a^p, b^r\}^u, \{b^s\}^v\} &= \{a^{u\cdot p}, b^{u\cdot r+v\cdot s}\}\\ \{a^p, b^r\} \times \{c^u\} &= \{(a,c)^{p\cdot u}, (b,c)^{r\cdot u}\}\end{aligned}$$

where $R \times S \triangleq \bigcup(x \in R) \bigcup(y \in S)\ (x, y)$.

We take the fact that the semantics of NRC_K is an instance of the general approach to collection languages promoted in e.g., [8, 21, 9] as evidence for the robustness of our semantics. Appendix A gives a set of equational axioms for NRC_K that follow from the general approach just mentioned. These axioms also form a foundation for query optimization for NRC_K and K-UXQuery (e.g., see [25]).

As positive *NRC* strictly extends the positive relational algebra (RA+), the following sanity check is also in order.

PROPOSITION 4. *Let NRC(RA+) be the usual encoding of projection, selection, cartesian product and union in (positive) NRC. The semantics of NRC(RA+) on K-complex values representing K-relations coincides with the semantics of RA+ on K-relations given in [16].*

As another sanity check, observe that $\mathit{NRC}_{\mathbb{N}}$ corresponds to the *positive* fragment of the *Nested Bag Calculus* [22].

Finally, we extend the semantics to $\mathit{NRC}_K + \mathit{srt}$. The semantics, given with respect to an environment to variables ρ, is summarized by the equations in Figure 8. The meaning of $\mathsf{Tree}(-,-)$, $\mathit{tag}(-)$ and $\mathit{kids}(-)$ are all straightforward (similar to pairing and projections). For srt, we require that Equation (1) continues to hold. Indeed, since K-collections have finite support, even in the presence of K-annotations, values of type *tree* have a finitary recursive structure. The semantics of $\bigcup(- \in -)\ -$ and Equation (1) above uniquely determines the semantics of srt.

6.3 Compiling UXQuery to $\mathit{NRC} + \mathit{srt}$

We define the semantics of K-UXQuery on K-UXML values by translation (compilation) to $\mathit{NRC}_K + \mathit{srt}$. Since K-UXML values can be expressed with the constructors in K-UXQuery it suffices to translate K-UXQuery. The compilation function is written $p \rightsquigarrow e$. Here we discuss some of the more interesting cases; more details can be found in the long version of this paper. Many of the operators in K-UXQuery have a direct analog in $\mathit{NRC}_K + \mathit{src}$ and therefore have a simple translation, for example if $p_1 \rightsquigarrow e_1$ and $p_2 \rightsquigarrow e_2$ then $\texttt{for}\ \$x\ \texttt{in}\ p_1\ \texttt{return}\ p_2 \rightsquigarrow \bigcup(x \in e_1)\ e_2$. The most interesting compilation rules concern navigation steps. The compilation of a step $ax\texttt{::}nt$, written $e \overset{ax\texttt{::}nt}{\rightsquigarrow} e'$, describes by e' the set of trees that results from applying the given step to the set of trees described by e. Navigation compilation is then used in query compilation: if $p \rightsquigarrow e$ and $e \overset{ax\texttt{::}nt}{\rightsquigarrow} e'$ then $p/ax\texttt{::}nt \rightsquigarrow e'$.

Here is an example of XPath compilation for the `self` axis combined with a node test a; it returns the trees whose root node is labeled by a:

$$e \overset{\texttt{self::}a}{\rightsquigarrow} \bigcup(x \in e)\ \mathit{if}\ \mathit{tag}(x) = a\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\}$$

The compilation of the `descendant` axis is the only place where we make use of structural recursion: we use srt to recursively walk down the structure of the tree and build up a set containing all of the matching nodes. As an example, the compilation rule for `descendant::*` is:

$$\frac{\begin{array}{l} e' = \bigcup(x \in e)\ \pi_1((\mathit{srt}(b, s).\ f)\ x)\\ \text{where } f = \mathit{let}\ \mathit{self} = \mathsf{Tree}(b, \bigcup(x \in s)\ \{\pi_2(x)\})\ \mathit{in}\\ \qquad\quad \mathit{let}\ \mathit{matches} = \bigcup(x \in s)\ \{\pi_1(x)\}\ \mathit{in}\\ \qquad\quad (\mathit{matches} \cup \{\mathit{self}\}, \mathit{self})\end{array}}{e \overset{\texttt{descendant::*}}{\rightsquigarrow} e'}$$

The s argument accumulates a set of pairs whose first component is the set of descendants below the immediate subtree contained in the second component of the pair. At each step, the body of the srt expression constructs a new pair using the current node and the accumulator. The descendants are obtained by projecting the first component of the final result.

6.4 Commutation with Homomorphisms

A semiring homomorphism $h : K_1 \xrightarrow{\text{hom}} K_2$ can be *lifted* to a transformation H from $\mathit{NRC}_{K_1} + \mathit{srt}$ expressions to $\mathit{NRC}_{K_2} + \mathit{srt}$ expressions by replacing every occurrence of a scalar k with $h(k)$. Since every K-complex value can be expressed with the constructors of $\mathit{NRC}_K + \mathit{srt}$, this gives us in particular a transformation from K_1-complex values to K_2-complex values.

A fundamental property of $\mathit{NRC} + \mathit{srt}$ is that query evaluation on K-complex values commutes with such transformations induced by homomorphisms.

THEOREM 1. *If $h : K_1 \xrightarrow{hom} K_2$ is a homomorphism of semirings, denote by H its lifting as explained above. Then for any K_1-complex value v and $\mathit{NRC}_{K_1} + \mathit{srt}$ query e, $H(e(v)) = H(e)(H(v))$.*

The proof is by induction on e.

In the same way, a homomorphism h can be lifted to a transformation H from K_1-UXQuery to K_2-UXQuery (and from K_1-UXML values to K_2-UXML values). Based on our compilation semantics for K-UXQuery, we conclude from the theorem above that a similar commutation holds for K-UXML and K-UXQuery:

COROLLARY 1. *If $h : K_1 \xrightarrow{hom} K_2$ is a semiring homomorphism, denote by H its lifting to a transformation from K_1-UXQuery to K_2-UXQuery. Then for any K_1-UXML value v and any K_1-UXQuery query p, $H(p(v)) = H(p)(H(v))$.*

We already mentioned several applications of the commutations with homomorphisms theorem (cf. §3, §4, and §5). Another simple but practically useful application involves the "duplicate elimination" homomorphism $\dagger : \mathbb{N} \to \mathbb{B}$ defined as $\dagger(0) \triangleq \textsf{false}$ and $\dagger(n + 1) \triangleq \textsf{true}$. Lifting $\dagger$ to K-complex values and trees or to K-UXML values we obtain that evaluation of ordinary values can be factored through that of values with multiplicities, with duplicate elimination deferred to a final step (in the style of commercial relational database systems).

7. SEMANTICS VIA RELATIONS

We sketch in this section an encoding of K-UXML into K-relations and an accompanying compilation of XPath into Datalog (extended with Skolem functions) which has the important property that the answer to the Datalog program corresponds to the answer to the XPath query with *identical annotations*. This provides an alternative definition of the semantics of XPath on K-UXML which agrees with that of §6. The availability of such a compilation scheme is an important concern in practice, where XML data is often "shredded" into relations, with queries over the data compiled into SQL for execution by an RDBMS [12, 28]. However, the focus here is not on practicality, but on demonstrating a basic proof-of-concept scheme.

We encode a (set of) K-UXML trees using a single K-relation $E(\mathit{pid}, \mathit{nid}, \mathit{label})$. Each tuple in E corresponds to a single K-UXML node, and carries the same annotation as the K-UXML node.

As opposed to UXML in which an item is an entire tree identified by its value, in this encoding an item is identified by its node id.

Thus *pid* is the identifier of the node's parent, *nid* is the identifier of the node itself, and *label* is the node's label. The special *pid* 0 is reserved and indicates that the node corresponds to a (top-level) root of a tree in the set.

Node ids are invented as needed during translation of the K-UXML into relational form. During subsequent query processing, additional node ids may be needed to represent nodes in the query result; we use Skolem functions for this purpose. Recursive Datalog rules are used to implement the XPath descendant operator. To give a flavor of the query translation, we show the rule for one important case, the `descendant` axis:

$$e \overset{\texttt{descendant::}a}{\rightsquigarrow} \quad \begin{array}{rcl} R(n,l) & \text{:-} & E(0,n,l) \\ R(n,l) & \text{:-} & R(p,\_), E(p,n,l) \\ E'(f(p),f(n),l) & \text{:-} & E(p,n,l) \\ E'(0,f(n),\mathsf{a}) & \text{:-} & R(n,\mathsf{a}) \end{array}$$

E encodes the set of input trees and E' encodes the set of output trees. f is a Skolem function. To illustrate, the XPath query `//c` on the source tree in Figure 4 with $x_1 := 0$ (to simplify the example) yields:

$$E' = \begin{array}{|ccc|l|} \hline \textit{pid} & \textit{nid} & \textit{label} & \\ \hline 0 & f(2) & c & y_1 \\ 0 & f(5) & c & y_1 \cdot y_2 \\ f(0) & f(1) & a & 1 \\ f(1) & f(2) & c & y_1 \\ f(2) & f(3) & d & 1 \\ f(3) & f(4) & a & 1 \\ f(2) & f(5) & c & y_2 \\ f(2) & f(6) & b & x_2 \\ \hline \end{array}$$

The K-UXML tree which would have been produced by executing the query directly on the input tree is encoded by the tuples reachable from the root tuples (which have *pid* 0). Note that there are also some "garbage" tuples in the table that are unreachable from any root: e.g., $(f(0), f(1), \mathsf{a})$. An additional step is required to remove these tuples; see [13] for details. We summarize with the following theorem:

THEOREM 2. *There is a 1-1 translation ϕ of K-UXML to K-relations and a translation ψ of XPath to Datalog with Skolem functions, such that for every K-UXML value v and XPath query p, we have $\phi(p(v)) = \psi(\phi(p))$.*

8. RELATED WORK

The original why/where provenance paper [7] actually used an XML-related data model. However, the model was tag-deterministic and the annotations were in effect paths from the root. Its query language relies on a deep-union construct that seems incomparable with what we do. In addition, the related work in [16] surveys work on semirings, other models of provenance, and probabilistic and incomplete relations that we do not repeat here.

Among proposed models for probabilistic and incomplete XML, closest to our work is [3, 27], which uses unordered XML decorated with Boolean combinations of probabilistic events. A model for incomplete XML was developed in [2]. In both systems the query language is tree patterns, and the main focus is on handling updates and complexity results. By contrast, our goal is a general-purpose annotation framework with a richer query language in which probabilistic and incomplete XML are obtained as special cases. Other models for probabilistic XML include probabilistic interval annotations [18], probabilistic trees for data integration [29], and numeric probability annotations [23]; for incomplete XML we add the maximal matchings approach of [20].

The focus of [5] is to compare a semantics for *NRC* on annotated complex values to the semantics of an update language but the data model and the query semantics is different from ours. In particular, query-constructed values are annotated with "unknown." Another provenance model for *NRC*, tracing operational executions for scientific dataflows, is described in [17].

Semirings are used to provide semantics for regular path queries decorated with preference annotations over graph-structured data in [14]. It is unclear whether there is any connection with our semiring-annotated *data*.

We note that, as in the conclusion to [16], we still don't know how to incorporate negative (more generally, non-monotonic) operations gracefully into this framework. Dealing with ordered XML is a separate but equally troublesome issue. Unlike sets and bags, lists are not immediately representable as the functions of finite support into some commutative (or even non-commutative) semiring. Still we believe that, based on our semantics for UXML, a practical, albeit somewhat ad-hoc, provenance semantics for ordered XQuery could be devised and then tested for user acceptance.

9. CONCLUSION AND FURTHER WORK

The framework for annotated XML we have described here seems to be flexible and potentially useful in practical applications. We are thinking in particular about using semirings of confidentiality levels in an RDBMS by hiding the out-of-model calculations from users and also about recording jointly provenance, security, and uncertainty (the product of several semirings is also a semiring!).

We have given very general strong representation systems in Section 5. This opens a whole set of questions about their (relative) completeness/expressive power. Another set of theoretical questions has to do with equivalence and perhaps containment wrt. annotated semantics, with applications to query optimization.

Acknowledgements We are grateful to Tova Milo for suggesting that we compare our first semantics with the "shredding" one. We also thank Zack Ives, Greg Karvounarakis, James Cheney, Jérôme Siméon, Giorgio Ghelli, and Kristoffer Rose for useful discussions, and the anonymous referees for many helpful comments. Our work is supported by the NSF under grants IIS-0534592, IIS-0447972, IIS-0629846 and IIS-05137782.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying xml with incomplete information. *TODS*, 31(1):208–254, 2006.

[3] S. Abiteboul and P. Senellart. Querying and updating probabilistic information in XML. In *EDBT*, 2006.

[4] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *PODS*, 2008.

[5] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT*, 2007.

[6] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[7] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[8] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *TCS*, 149(1):3–48, 1995.

[9] P. Buneman and V. Tannen. A structural approach to query language design. In *The Functional Approach to Data Management Modeling, Analyzing, and Integrating Heterogenous Data*. Springer, 2004.

[10] J. V. den Bussche, D. V. Gucht, and S. Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus and XQuery. In *ICDT*, 2005.

[11] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C, Jan. 2007.

[12] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[13] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and provenance. Technical Report TR-CIS-08-06, University of Pennsylvania, 2008.

[14] G. Grahne, A. Thomo, and W. W. Wadge. Preferentially annotated regular path queries. In *ICDT*, 2007.

[15] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[16] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[17] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. A formal model of dataflow repositories. In *DILS*, 2007.

[18] E. Hung, L. Getoor, and V. S. Subrahmanian. Probabilistic interval XML. *ACM TOCL*, 8(4), 2007.

[19] T. Imieliński and W. Lipski. Incomplete information in relational databases. *JACM*, 31(4), 1984.

[20] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with incomplete answers over semistructured data. In *PODS*, 1999.

[21] S. K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science*, 1997.

[22] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *JCSS*, 55(2):241–272, 1997.

[23] A. Nierman and H. V. Jagadish. ProTDB: Probabilistic data in XML. In *VLDB*, 2002.

[24] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT Workshops*, 2002.

[25] C. Ré, J. Simèon, and M. Fernández. A complete and efficient algebraic compiler for xquery. In *ICDE*, page 14, 2006.

[26] E. L. Robertson, L. V. Saxton, D. V. Gucht, and S. Vansummeren. Structural recursion on ordered trees and list-based complex objects. In *ICDT*, 2007.

[27] P. Senellart and S. Abiteboul. On the complexity of managing probabilistic XML data. In *PODS*, 2007.

[28] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB J.*, 1999.

[29] M. van Keulen, A. de Keijzer, and W. Alink. A probabilistic XML approach to data integration. In *ICDE*, 2005.

APPENDIX

A. MONADS OF SEMIMODULES AS COLLECTION TYPES

Let $(K, +, \cdot, 0, 1)$ be a commutative semiring. A *semimodule* over K (a K-semimodule) is an algebraic structure $(M, +, 0, \lambda)$ where $(M, +, 0)$ is a commutative monoid, and $\lambda : K \times M \to M$ is a *scalar multiplication operation*, written (as usual) $\lambda(k, x) = k\,x$ such that

$$\begin{aligned} k\,(x+y) &= k\,x + k\,y \\ k\,0 &= 0 \\ (k_1 + k_2)\,x &= k_1\,x + k_2\,x \\ (k_1 \cdot k_2)\,x &= k_1\,(k_2\,x) \\ 0\,x &= 0 \\ 1\,x &= x \end{aligned}$$

K-semimodules and their homomorphisms form a category K-**SMod**. The forgetful functor U :$\mathrm{K}{-}\mathbf{SMod} \to \mathbf{Set}$ has a left adjoint that is very easy to describe: the free K-semimodule generated by a set X is the set X_f^K of functions $X \to K$ that have *finite support* (see Section 6.2 and note that in *NRC*$_K$ $[\![\{t\}]\!]_K$ is precisely $([\![t]\!]_K)_f^K$), with the obvious pointwise addition and pointwise multiplication K-semimodule structure. This adjunction yields a (strong) monad on **Set**, which can be enriched [21] with a K-semimodule structure on each monad algebra. Therefore, we have a collection and aggregates query language, as in [8, 21, 9]. In fact, it is easy to see that any commutative monoid is an $\mathbb{N}$-semimodule and that the $\mathbb{B}$-semimodules are exactly the commutative-idempotent monoids, so the finite sets and finite bags collections are included here [11]. Properties like the commutation with homomorphisms theorem (1) have a very general category-theoretic justification, based on the fact that all the query language constructs in such query languages come from functorial constructs and natural transformations.

We can also capture some of this theory through an equational axiomatization for *NRC*$_K$

PROPOSITION 5. *The semantics of NRC*$_K$ *satisfies the following equational axioms:*

- $\cup$, $\{\}$ *and multiplication with scalars from* K *satisfy the axioms of a* semimodule *over* K.
- $\bigcup(x \in e_1)\; e_2$ *satisfies the axioms:*

$$\begin{aligned} \textstyle\bigcup(x \in \bigcup(y \in R)\; S)\; T &= \textstyle\bigcup(y \in R) \bigcup(x \in S)\; T \\ \textstyle\bigcup(x \in S)\; \{x\} &= S \\ \textstyle\bigcup(x \in \{e\})\; S &= S[x := e] \\ \textstyle\bigcup(x \in k_1\, R_1 \cup k_2\, R_2)\; S &= k_1\, (\textstyle\bigcup(x \in R_1)\; S) \;\cup \\ &\quad\; k_2\, (\textstyle\bigcup(x \in R_2)\; S) \\ \textstyle\bigcup(x \in R) \bigcup(y \in S)\; T &= \textstyle\bigcup(y \in S) \bigcup(x \in R)\; T \\ \textstyle\bigcup(x \in R)\; (k_1\, S_1 \cup k_2\, S_2) &= k_1\, (\textstyle\bigcup(x \in R)\; S_1) \;\cup \\ &\quad\; k_2\, (\textstyle\bigcup(x \in R)\; S_2) \end{aligned}$$

(In particular, the 4th and 6th axioms state its bilinearity *w.r.t. the semimodule structure).*

[11] It is not clear how to include finite lists in this semiring-based family of collection types