June 2008

**Annotated XML: Queries and Provenance**

John N. Foster  
*University of Pennsylvania, jnfoster@cis.upenn.edu*

Todd J. Green  
*University of Pennsylvania, tjgreen@cis.upenn.edu*

Val Tannen  
*University of Pennsylvania, val@cis.upenn.edu*

Follow this and additional works at: [https://repository.upenn.edu/db_research](https://repository.upenn.edu/db_research)

---

Publisher URL: [http://doi.acm.org/10.1145/1376916.1376954](http://doi.acm.org/10.1145/1376916.1376954)

This paper is posted at ScholarlyCommons. [https://repository.upenn.edu/db_research/41](https://repository.upenn.edu/db_research/41)  
For more information, please contact repository@pobox.upenn.edu.
Annotated XML: Queries and Provenance

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.

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

Comments
Publisher URL: http://doi.acm.org/10.1145/1376916.1376954

This conference paper is available at ScholarlyCommons: https://repository.upenn.edu/db_research/41
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], we focused on an unordered variant of XML, which does not seem to be related to the way we use semirings (see §8), 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 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), while the second uses an encoding that

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 \( O(|D|^{|0|}) \) 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 future work in §9. The long version of this abstract contains the

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

- \( (\mathbb{B}, \lor, \land, \text{false, true}) \): 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 and, in fact, capture the generality of all commutative semiring calculations: any function \( X \to K \) can be uniquely extended to a semiring homomorphism \( \mathbb{N}[X] \to K \). This fact is relevant to querying since (as in [16]) by Theorem 1 and Corollary 1 below, our semantics for querying 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 \( L \) of labels, the usual mutually recursive definition of XML data naturally generalizes to \( K \)-UXML:\(^4\)

- A value is either a label in \( L \), a tree, or a \( K \)-set of trees;
- A tree consists of a label together with a (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 \( (\mathbb{B}, \lor, \land, \text{false, true}) \) 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

\(^3\)These polynomials can be used, for example, to track provenance in systems for scientific data sharing, see [15].

\(^4\)For simplicity, we also omit attributes and model atomic values as the labels on trees having no children.

\(^5\)In the XQuery data model, sets of labels are also values; it is straightforward to extend our formal treatment to include this.

\(^6\)Items annotated with 0 are allowed by the definition but are useless because our semantics interprets 0 as “not present/available”.

\[^2\] The product \( k_1 \cdot k_2 \) is the XQuery program that defines the

\[^1\] The sum \( k_1 + k_2 \) is the XQuery program that defines the

\[^3\] The minimum \( \min(k_1, k_2) \) is the XQuery program that defines the

\[^4\] The maximum \( \max(k_1, k_2) \) is the XQuery program that defines the

\[^5\] The logical product \( k_1 \land k_2 \) is the XQuery program that defines the

\[^6\] The logical sum \( k_1 \lor k_2 \) is the XQuery program that defines the

\[^7\] The existential quantifier \( \exists k \) is the XQuery program that defines the

\[^8\] The universal quantifier \( \forall k \) is the XQuery program that defines the

\[^9\] The implication \( k_1 \rightarrow k_2 \) is the XQuery program that defines the

\[^10\] The equivalence \( k_1 \equiv k_2 \) is the XQuery program that defines the

\[^11\] The disjunction \( k_1 \lor k_2 \) is the XQuery program that defines the

\[^12\] The conjunction \( k_1 \land k_2 \) is the XQuery program that defines the

\[^13\] The complement \( \neg k \) is the XQuery program that defines the

\[^14\] The negation \( \sim k \) is the XQuery program that defines the

\[^15\] The existential quantifier \( 

XML

\[
\begin{pmatrix}
\begin{array}{c}
\text{Source:}
\
\text{Answer:}
\\
\text{Figure 1: Simple Example.}
\\
l \in \mathcal{L}
\\k \in \mathcal{K}
\\p ::= l | \{x\} | (p) | p \cdot p
\\| \text{let } \{x\} \text{ return } p
\\| \text{if } (p = p) \text{ then } p \text{ else } p
\\s ::= \text{ax | child | descendant}
\\nt ::= l \\
\end{array}
\end{pmatrix}
\]

where we have abbreviated leaves \( <l> </l> \) 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 \( \text{annot} \) allows queries to modify the annotations on sets. With \( \text{annot} k \), any K-UXML value can be built with the K-UXQuery constructs.

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

\[
t ::= \text{label} \mid \text{tree} \mid \{\}\n\]

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

\[
\begin{pmatrix}
b \&\& a^{k_1 \cdot k_2} \\
\end{pmatrix}
\]

Next, let us examine a query that uses iteration:

\[
p = \text{element } p \{\text{for } \{x\} \in \{\}\text{ return for } \{x\} \in (\{\})/\text{ return } (\{x\}/*)\}
\]

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. Operationally, the query works as follows. First, the outer \( \text{for}-\text{clause} \) 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 \( \text{for}-\text{clause} \) iterates over the set of trees given by \( (\{t\}/*) \).

\[
\Gamma \vdash p_1 : \{\text{tree}\} \quad \Gamma \vdash p_2 : \{\text{tree}\}
\]

\[
\Gamma \vdash p_1 : \{\text{tree}\} \quad \Gamma, x : \text{tree} \vdash p_2 : \{\text{tree}\}
\]

\[
\Gamma \vdash \text{for } \{x\} \in p_1 \text{ return } p_2 : \{\text{tree}\}
\]

\[
\Gamma \vdash p_1 : \{\text{label}\} \quad \Gamma \vdash p_2 : \{\text{label}\} \quad \Gamma \vdash p_3 : t \quad \Gamma \vdash p_4 : t
\]

\[
\Gamma \vdash \text{if } (p_1 \neq p_2) \text{ then } p_3 \text{ else } p_4 : t
\]

\[
\Gamma \vdash p_1 : \{\text{label}\} \quad \Gamma \vdash p_2 : \{\text{label}\} \quad \Gamma \vdash p_3 : t \quad \Gamma \vdash p_4 : t
\]

\[
\Gamma \vdash \text{element } p_1 \{p_2\} : \{\text{tree}\}
\]

\[
\Gamma \vdash \text{name}(p_1) : \{\text{label}\}
\]

\[
\Gamma \vdash p : \{\text{tree}\}
\]

\[
\Gamma \vdash k : \mathcal{K}
\]

\[
\Gamma \vdash p/\text{ax :: nt} : \{\text{tree}\}
\]

\[
\Gamma \vdash \text{annot } k \ : \{\text{tree}\}
\]

It binds \( \{x\} \) to each of these trees, evaluates the \( \text{return-clause} \) 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 \( \text{return-clause} \) produces the singleton set \( \{d^{y_1}\} \). Multiplying this set by the annotation \( x_1 \) yields \( \{d^{y_1 \cdot y_1}\} \). After combining all the sets returned by iterations of this inner \( \text{for}-\text{clause} \), we obtain the set \( \{d^{y_1 \cdot y_2}, d^{y_2 \cdot y_3}\} \). 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

\[
r = \text{element } r \{\{t\}/c\}
\]

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

\[\text{Actually this query is equivalent to the shorter "grandchildren" XPath query } \{\{t\}/c\}/\ast \text{; we use the version with a } \text{for-clauses} \text{ to illustrate the semantics of iteration.}\]
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 $R(A, B, C)$ and $S(B, C)$ shown at the top of Figure 5. The result, cf. [16], is the $K$-relation $Q(A, C)$, also shown at the top of Figure 5. For example, the annotation on (d e) 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 $[e]$.)

Proposition 1. Let $Q$ 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 subterms which were distinguished by annotations become now identified (q₁ and q₂, q₄ and q₅); 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 UXQuery value annotated with indeterminates from a set 𝑋 and 𝑝 is a UXQuery, then computing 𝑝(𝑣) according to the °[𝑋]-UXQuery semantics produces an °[𝑋]-UXQuery value such that the size of any of the provenance polynomials that annotate 𝑝(𝑣) is 𝑂(|𝑣|⁹).

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 $x/z/a$ above, not $(x/z)/a$.

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 $C: P < C < S < T < 0$ describes the following levels of “clearance”: $P =$ public, $C =$ confidential, $S =$ secret, and $T =$ top-secret. It is easy to see that $(C, \min, \max, 0, P)$ 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 $C$ as follows $w_i := C$ (the entire relation

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

![Figure 7: Security Clearance Example.](image)

R denotes confidentiality, $x_2 := S$ (in addition, this tuple is secret), and $y_2 := T$ (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).

The result of the view/query in Figure 5 when applied to this data is the $C$-UXML encoding of a relation in which only the annotations on the tuples are different from $1 = P$ (this is because the query projects out the attribute $B$, otherwise we could have had non-⁴ 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 $N[w_1, x_2, y_2]$. 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 $C$ under the valuation $w_1 := C, x_2 := S, y_2 := T$ we get the same result as the 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 $B$ 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 $B$-UXML) database. For repetitions (multiplicities) the possible worlds are $N$-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 $N[X, +, \cdot, 0, 1]$—i.e. polynomial expressions over variables $X$ with natural number coefficients [16]. For any commutative semiring $K$, provenance polynomials are “universal”
in the sense that any function \( f : X \rightarrow K \) (we call \( f \) a valuation) extends uniquely to a semiring homomorphism \( f^* : \mathbb{N}[X] \rightarrow K \).

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 \( \text{Mod}_K \) 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., \( \text{Mod}_K(v) \) is \( \{ f^*(v) : f : X \rightarrow K \} \), 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:

\[
\text{Mod}_\mathbb{B}(v) = \left\{ \begin{array}{c}
\begin{array}{cccccccc}
a & a & a & a & a & a & a & a \\
b & c & b & b & b & b & b & b \\
a & d & c & b & a & a & d & b \\
\end{array}
\end{array} \right\}
\]

Each tree in \( \text{Mod}_\mathbb{B}(v) \) is obtained using a valuation from the \( y_1 \to \mathbb{B} \)---e.g., for the rightmost tree in this display, the valuation maps \( y_1 \) to \text{true} and \( y_2 \) and \( y_3 \) to \text{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., \( p(\text{Mod}_K(v)) = \{ p(v') : v' \in \text{Mod}_K(v) \} \).

Returning to the representation \( v \) above and using \( p \), the query in Figure 4, we have:

\[
p(\text{Mod}_\mathbb{B}(v)) = \left\{ \begin{array}{c}
\begin{array}{cccccccc}
& & & & & & & \\
& & & & & & & \\
& & & & & & & \\
& & & & & & & \\
\end{array}
\end{array} \right\}
\]

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 \( p(\text{Mod}_K(v)) = \text{Mod}_\mathbb{B}(p(v)) \).

Indeed, the specific answer for this example shown in Figure 4 is the representation of \( p(\text{Mod}_\mathbb{B}(v)) \). Using the terminology of incomplete databases, we say that \( \mathbb{N}[X] \)-UXML is a strong representation system for \( \mathbb{B} \)-UXQuery and \( \mathbb{B} \)-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 semantic \( \text{PosBool}(B, \lor, \land, \text{false}, \text{true}) \) of positive Boolean expressions over a set \( B \) of variables (i.e., the expressions involve only \( B \), disjunction, conjunction, and constants for \( \text{true} \) and \( \text{false} \)). This corresponds to an XML analogue of the Boolean \( c \)-tables [19] used in incomplete databases. Valuations \( v : B \rightarrow \mathbb{B} \) extend uniquely to homomorphisms \( v^* : \text{PosBool}(B) \rightarrow \mathbb{B} \), so the definition above of \( \text{PosBool} \) still makes sense. Indeed, it follows (again from the commutation with homomorphisms in \$6.4\) that \( \text{PosBool}(B) \)-UXML is a strong representation system for \( \text{UXQuery} \) and \( B \)-UXML (i.e., ordinary UXML) and that we can transform an \( \mathbb{N}[B] \)-UXML representation into \( \text{PosBool}(B) \)-UXML representation by applying the obvious homomorphism. It can be shown that \( \text{PosBool} \) 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:

\[
\text{Mod}_\mathbb{N}(v) = \left\{ \begin{array}{cccccccc}
a & a & a & a & a & a & a & a \\
b & b & b & b & b & b & b & b \\
a & a & a & a & a & a & a & a \\
\end{array} \right\}
\]

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 \( f : X \rightarrow K \). 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 \( \text{Pr}[f(x) = n] = 1/2^n \) 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 \( \text{PosBool} \) 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 \( \text{NRC} \) 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. 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 \( \text{NRC} \) are:

\[
t ::= \text{label} \mid t \times t \mid \{t\}
\]

Complex values are built with the following constructors:

\[
v ::= l \mid (v, v) \mid \{v\} \mid v \cup v
\]

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 \( \text{label} \times \{\text{label}\} \).

---

\(^8\)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

\[
\text{project}_1 r \triangleq \bigcup(x \in R) \{\pi_1(x)\}.
\]

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

\[
\Gamma \vdash v_1 : \text{label} \quad \Gamma \vdash v_2 : \{\text{tree}\} \\
\Gamma \vdash \text{Tree}(v_1, v_2) : \text{tree}
\]

where \( \text{tree} \) is a new type. It is easy to see that the values of type \( \text{tree} \) and \( \text{label} \times \{\text{tree}\} \) are in a 1-1 correspondence. In one direction this isomorphism is witnessed by \( \text{Tree}(\pi_1(P), \pi_2(P)) \), where \( P \) is a pair. To express the other direction, we extend the calculus with two new operations, \( \text{tag}(-) \) and \( \text{kids}(-) \) that return the root tag and the set of subtre children of the root, respectively. The mapping from trees to pairs is then given by \( (\text{tag}(T), \text{kids}(T)) \), 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:

\[
\Gamma, x : \text{label}, y : \{\} \vdash e_1 : t \\
\Gamma \vdash (\text{srt}(x, y), e_1) e_2 : t
\]

Its semantics obeys the equation

\[
(\text{srt}(x, y), e_1) \text{Tree}(e_2, e_3) = e_1[x := e_2, y := \bigcup(z \in e_3) (\text{srt}(x, y), e_1) z]
\]

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

\[
(\text{srt}(x, y), \{x\} \cup \text{flatten } y) t
\]

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

6.2 Semantics for \( \text{NRC + srt} \)

Next we show how to decorate complex values (and trees) with semiring annotations, and generalize \( \text{NRC + srt} \) 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 \( \{\} \). The usual semantics is the set of finite subsets of \( \{\} \). Instead, the semantics of \( \{(\{\})K \) is defined as the set of functions \( f : \{(\{)K \to K \) with finite support, i.e., such that \( \text{supp}(f) := \{a \in \{(\{)K | f(a) \neq 0\} \) is finite. We call elements of \( \{(\{)K \) K-complex values. 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 Kollections. We define new semantics for the NRC constructors: the singleton constructor \( \{(v)K \) is the function that maps \( vK \) 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:

\[
\Gamma \vdash k \in K \\
\Gamma \vdash e : \{t\} \\
\Gamma \vdash k e : \{t\}
\]

We call the calculus extended with this operator \( \text{NRC}_K \). The set of K-complex values are constructed using:

\[
v \ :: := l \ | (v, v) \ | k \{v\} \ | v \cup v \ | \{\}
\]

and, as above, we abbreviate K-collections using the following notation: \( \{e_1 \cup \cdots \cup e_n\}K \triangleq k_1 \{e_1\}K \cup \cdots \cup k_n \{e_n\}K \). 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 \( \text{supp}(f) \) \( \subseteq \{t_1 \} \). In general \( e_2 \) depends on \( x \) so for each \( x \) we have a corresponding semantics for \( e_2 \), i.e., a function \( g_x : Y \to K \). Using this function we define for each \( y \in Y \)

\[
[\bigcup(x \in e_1) e_2]\rangle K (y) \triangleq \sum_{i=1}^{n} f(x_i) \cdot g_x (y)
\]

Since each \( g_x \) has finite support, so does \( [\bigcup(x \in e_1) e_2]\rangle K \).

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
example,
\[
\text{flatten } \{\{a^p, b^r\}, \{b^s\}\} = \{a^{u+p}, b^{r+v+s}\}
\]
\[
\{a^p, b^r\} \times \{c^t\} = \{(a, c^{p+u}), (b, c)^{r+v}\}
\]
where \(R \times S \triangleq \bigcup(x \in R) \bigcup(y \in S) (x, y)\).

We take the fact that the semantics of NRC(\(RA^+\)) 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(\(RA^+\)) 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 NRC(\(K\)) corresponds to the positive fragment of the Nested Bag Calculus [22].

Finally, we extend the semantics to NRC(\(RA^+\)) + srt. The semantics, given with respect to an environment to variables \(\rho\), is summarized by the equations in Figure 8. The meaning of Tree(\(-, -\)), tag(\(-\)) and 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 \(\bigcup(\varepsilon \in \varepsilon)\) – and Equation (1) above – uniquely determines the semantics of srt.

6.3 Compiling UXQuery to NRC + srt

We define the semantics of K-UXQuery on K-UXML values by translation (compilation) to NRC(\(K\)) + 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 \(p \Rightarrow e\). 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 NRC\((K\) + srt) and therefore have a simple translation, for example if \(p_1 \Rightarrow e_1\) and \(p_2 \Rightarrow e_2\) then for \(x \in p_1\) return \(p_2 \Rightarrow (x \in e_2)\). The most interesting compilation rules concern navigation steps. The compilation of a step \(ax::nt\), written \(e \xrightarrow{ax::nt} e'\), 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 \(p \Rightarrow e\) and \(e \xrightarrow{ax::nt} e'\) then \(p/ax::nt \Rightarrow e'\).

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 \(a\):

\(e \xrightarrow{\text{self}/\text{nt}} \bigcup(x \in e) \text{ if } \text{tag}(x) = a \text{ then } \{x\} \text{ else } \{\}\)

The composition 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:

\(e' = \bigcup(x \in e) \tau((srt(b, s), f) x)\)

where \(f = \text{let self} = \text{Tree}(b, \bigcup(x \in s) \{\tau_2(x)\})\) in

\(\text{let matches} = \bigcup(x \in e) \{\tau_1(x)\}\) in

\((\text{matches} \cup \{\text{self}\}, \text{self})\)

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 \(h : K_1 \rightarrow K_2\) can be lifted to a transformation \(H\) from NRC\((K_1, srt)\) expressions to NRC\((K_2, srt)\) expressions by replacing every occurrence of a scalar "\(k\) with \(h(k)\). Since every K-complex value can be expressed with the constructors of NRC\((K\) + srt), this gives us in particular a transformation from K-complex values to K-complex values.

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

**Theorem 1.** If \(h : K_1 \rightarrow K_2\) is a homomorphism of semirings, then for any \(K_1\)-complex value \(v\) and NRC\((K_1, srt)\) query \(e, H(v(e(h(v)))) = H(v)H(v)\).

The proof is by induction on \(e\).

In the same way, a homomorphism \(h\) can be lifted to a transformation \(H\) from K-UXQuery to K-UXQuery for K-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 \(h : K_1 \rightarrow K_2\) is a semiring homomorphism, then for any K-UXML value \(v\) and K-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 \(\downarrow : N \rightarrow 2^k\) defined as \(\downarrow(0) = \text{false}\) and \(\downarrow(n+1) = \text{true}\). Lifting \(\downarrow\) 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(pid, nid, 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 Datatalog 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:

$$R(n, l) \leftarrow E(0, n, l)$$

$$R(n, l) \leftarrow R(p, _), E(p, n, l)$$

$$E'(f(p), f(n), l) \leftarrow E(p, n, l)$$

$$E'(0, f(n), a) \leftarrow R(n, a)$$

$E$ 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:

$$E' = \begin{array}{|c|c|c|}
\hline
pid & nid & label \\
\hline
0 & f(2) & c & y_1 \\
0 & f(5) & c & y_1 \cdot y_2 \\
0 & f(1) & a & 1 \\
1 & f(2) & c & y_1 \\
1 & f(3) & d & 1 \\
1 & f(4) & a & 1 \\
1 & f(5) & c & y_2 \\
1 & f(6) & b & z_2 \\
\hline
\end{array}$$

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., $(f(0), f(1), a)$. 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 $\phi$ of $K$-UXML to $K$-relations and a translation $\psi$ 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


**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) $\lambda(k, x) = k \cdot x$ such that

\[
\begin{align*}
k(x + y) &= kx + ky \\
k0 &= 0 \\
(k_1 + k_2)x &= k_1x + k_2x \\
(k_1 \cdot k_2)x &= k_1(k_2x) \\
0x &= 0 \\
1x &= x
\end{align*}
\]

$K$-semimodules and their homomorphisms form a category $K\text{-Semimod}$. The forgetful functor $U : K\text{-Semimod} \to \text{Set}$ has a left adjoint that is very easy to describe: the free $K$-semimodule generated by a set $X$ is the set $X^F$ of functions $X \to K$ that have *finite support* (see Section 6.2 and note that in $NRC_K$, $[[t]]_K$ is precisely $([t])^K$), with the obvious pointwise addition and pointwise multiplication $K$-semimodule structure. This adjunction yields a (strong) monad on $\text{Set}$, 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 $\mathbb{N}$-semimodules are exactly the commutative-idempotent monoids, so the finite sets and finite bags collections are included here.

Properties like the commutative 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:

- $U \cup \{ \} \text{ and multiplication with scalars from } K \text{ satisfy the axioms of a semimodule over } K.$
- $U(x, e_1) \cup e_2 \text{ satisfies the axioms:}
  \begin{align*}
  U(x \in U(y \in R) S) T &= U(y \in R) \cup (x \in S) T \\
  U(x \in S) \cup U(x \in \{e_1\}) &= S \cup U(x \in \{e_1\}) \\
  U(x \in k_1 R_1 \cup k_2 R_2) S &= k_1 U(x \in R_1) S \cup k_2 U(x \in R_2) S \\
  U(x \in R) \cup (y \in S) T &= U(y \in S) \cup U(x \in R) T \\
  U(x \in R) \cup (k_1 S_1 \cup k_2 S_2) &= k_1 U(x \in R) S_1 \cup k_2 U(x \in R) S_2.
  \end{align*}

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

---

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