A deterministic model for semi-structured data

Peter Buneman  Alin Deutsch  Wang-Chiew Tan
University of Pennsylvania
Email: peter@cis.upenn.edu

1 Introduction

This is a preliminary report on a new model for semi-structured data. The idea of semi-structured data evolved, in part, from various syntactic representations of data such as AceDB[11], OEM[10] and has more recently been used effectively to design query languages for XML[4]. Insofar as there is an agreed model, it is simply an edge-labeled graph. However this description begs a number of important questions: What can constitute an edge label? Are there values associated with the vertices? Is there a separate labeling system for vertices to provide them with independent identity?

We describe here a new model for semi-structured data. It is more restrictive than the models described in [10, 6, 5] in that it is deterministic. The edges emanating from any node in the graph have distinct labels. It is less restrictive in that the edges can carry data and may have structure. In fact they may themselves be small pieces of semi-structured data. The advantage of this approach is that each component of the database is uniquely identified by a path. Paths not only serve as object identifiers; but unlike object identifiers, paths also have structure, and a number of useful database operations may be obtained by manipulation of this structure.

The motivation for this model comes from a number of structures that we would like to consider as databases. AceDB, for example, allows edges whose labels may be integers or strings. It is common practice to use file systems as databases, and it often happens that the file names themselves carry data. For example, the following description is to be found in a linguistic database file structure[4].

/timit/train/drt/fcjfo/sa1.wav

(TIMT corpus, training set, dialect region 1, female speaker, speaker-ID "cjfo", sentence text "sat", speech waveform file)

Notice that some of these file names/edge labels are themselves identifiers and may be used elsewhere as paths for example, we might expect files such as /speakers/cjfo0.txt and /sentences/sa1.txt to be available.

One of the goals of this work is to be able to describe an annotation system for databases; annotations are commonly needed for describing where data came from, corrections, exceptions etc. While a database may conform to a schema, the annotations are necessarily irregular and it is appropriate to use a semi-structured model to describe such annotations. However, we have found we need a deterministic form of semi-structured data to convey annotations.

In this paper we describe the data model and the syntax and semantics of constructing data; we describe constraints and fundamental operations on data. We then describe a type system that may be used to form views of data and finally address the issue of query languages. In this model the well-definedness of queries is important, and we provide some preliminary results on conditions that guarantee the well-definedness of queries.

2 Syntax

Definitions of semi-structured data start from an explicit syntax for data. We shall adopt a syntax close to that developed for OEM and UnQL[1]. The syntax in these models describes the value associated with a vertex in the graph, which is a set of label/value pairs. We use the notation \( x : y \) to denote a pair whose label is \( x \) and value is \( y \) and the notation \( \{ x_1 : y_1, \ldots, x_n : y_n \} \) for a set of such pairs.

Examples:

1. A record: \{Name: "Bruce", Height: 6.2\}
3. A set: \{1, 2, \{\}\, 4: \{\}\}
4. A multiset: \{"Guinness": 3, "Nelson": 2\}
5. A relation with an explicit key:

\{\{Id: 11\}: \{Name: "Kim", Rate: 60\},
{Id: 22}: \{Name: "Bob", Rate: 75\}\}

What we see in these examples is simply a relaxation of the conventional syntax for semi-structured data as expressed in say OEM. The edge labels are no longer restricted to being drawn from some syntactic class that is reserved for edge labels; they may be arbitrary values and may themselves be pieces of semi-structured data. However these examples also satisfy the restriction that in any expression \( \{ x_1 : y_1, \ldots, x_n : y_n \} \) the labels \( x_1, \ldots, x_n \) are all distinct, so that our syntax represents a finite partial function or map from values to values. The examples above all appear to conform to some type system, however there is nothing to prevent us writing "unstructured" expressions such as \( \{1: "a", 2: {}\}: age\).

Semantics. Although the semantics of our constructs follows almost trivially from the syntax, it is worth describing it here in order to introduce a little terminology.

---

We could also cast this exposition in XML, however the added verbosity and the fact that XML is an open question as to what should be represented be attributes or PC data.
Semantically, a map from a set $X$ to a set $Y$ is simply a finite subset $m$ of $X \times Y$ such that if $(x, y_1) \in m$ and $(x, y_2) \in m$, then $y_1 = y_2$. The domain of $m$ $\text{dom}(m)$ is the set $\{x \mid \exists y((x, y) \in m)\}$. The image of $m$ is similarly defined. We can describe an arbitrary set as a map by pairing each of its members with some arbitrary value such as the empty map. This is what we did in example 3 above. Thus the semantics of our data model is so far described as the least fixed point the equation $V \cong B + V - \text{fin} V$.

**Paths.** The advantage of using a model based on maps is that a sequence of values uniquely identifies a component of a one of our structures. For example, \{Id: 11\} identifies a a tuple, \{Id: 11\}, Rate identifies the value 50. We will call a sequence of values a path, and we shall augment our space of values with paths which, in our syntax, we shall write $(v_1, v_2, \ldots, v_n)$. Example:

```
{Emps: {Id: 11}: {Name: "Kim", Rate: 50},
 Id: 22}: {Name: "Bob", Rate: 75, 
 Manager = <Emps.Id: 11>}} } 
```

The semantics of values is now augmented with paths to obtain the least fixed point of $V \cong B + V - \text{fin} V + \text{seq}(V)$ in which $\text{seq}(V)$ is the set of finite sequences of values.

Clearly, there are constraints that should be imposed on such expressions. Before discussing them we introduce some additional syntactic sugar. The disjoint union $m_1 + m_2$ of two maps is $m_1 \cup m_2$ provided $\text{dom}(m_1) \cap \text{dom}(m_2) = \emptyset$. Using this we shall extend our syntax with the notation $e_1 : + e_2$. Thus \{Id: 11\}: {Name: "Kim"} is shorthand for \{Id: 11\}, {Name: "Kim"} and \{Id: 11\}. Id is now a valid path in this schema.

A second notation we shall introduce is one for path labels. By placing a label before a pair we are make it an alias for the path to that pair. For example, we can write our example above as

```
{Emps: { #e1{Id: 11}: {Name: "Kim", Rate: 50}, 
 #e2{Id: 22}: {Name: "Bob", Rate: 75, 
 Manager = #e1} } } 
```

Using these ideas we can construct the simple database in figure 1.

It should be emphasized that path labels are just that. While they look like object identifiers, it is the paths they denote that are the object identifiers. This is the crucial difference between this model and the OEM or UnQL models. Note that the appearance of a path in the “key” of an object, as in \{Emp: #e1, Client: #c1\} indicates an object dependency.

It should be apparent that this syntax is extremely close to that of data formats such as OEM and other formats, and could easily represented in XML. With a few modifications languages for these formats can be applied directly to this model. However we shall be concerned with new operations that interact with constraints and produce maps rather than sets as outputs.

### 3 Constraints and operations.

We start by formulating some obvious constraints on values. We observe that equality of values can be inductively checked and therefore the constraint on our syntax that expressions represent partial functions can also be checked.

A path $p$ occurs in a value $v$ if $p = v$, or $v$ is a map and $p$ occurs in an element of $\text{dom}(v) \cup \text{codomain}(v)$, or $v$ is a path and $p$ occurs in an element of $p$.

By an extension of application of a value to another value, if $p$ is a path we define $v(p)$ to be $v$ if $p$ is empty, and to be $v(v'(p'))$ if $p = v' \cdot p'$ and $v'$ is defined. Otherwise it is undefined. Note that this definition does not “dereference” paths.

A path $p$ is an identifier for $v$ if $v(p)$ is defined.

The path table of a value $v$ is the set of pairs of paths $(p, p')$ such that $p' = v(p)$.

For example, the path table for the database in figure 1 is the binary relation

```
... #e1{Id: 1}: ... <Emps.Id: 22>.Manager <Emps.Id: 11> <Emps.Id: 32>.Manager <Emps.Id: 11> 
```
**Definition 3.1** A value \( v \) is **consistent** if every path that occurs in \( v \) is an identifier for \( v \) and if the path table is acyclic.

The first condition states that every path that is mentioned in \( v \) must be a path that can be used to access a part of \( v \). The second condition avoids constructs such as \( \{B:C\}, \{A:B\} \) in which we have paths that “point to” themselves. This ensures that every sub-expression of an expression has a unique canonical identifier.

**Deep operations.** Just as the notion of applying a value to another value can be generalized to a “deep” application – applying a value to a path, there are other operations that can be generalized to “deep” versions. Unlike (disjoint) union, the deep union of \( m_1 \) and \( m_2 \) does not require their domains to be disjoint.

**Definition 3.2** (Deep Union)

Deep union \( (m_1, m_2) \) =

\[
\{ (x_1, y_1), (x_2, y_2) \mid (x_1, y_1) \in m_1 \land x_1 \notin \text{dom}(m_2) \} \cup \\
\{ (x_2, y_2), (x_3, y_3) \in m_2 \land x_2 \notin \text{dom}(m_1) \} \cup \\
\{ (x, y_1), (x, y_2) \mid (x, y_1) \in m_1 \land (x, y_2) \in m_2 \}
\]

where \( g(y_1, y_2) = \)

- deepu \( (y_1, y_2) \) if \( y_1 \) and \( y_2 \) are both maps,
- undefined otherwise

Example: The deep union of \( \{A:1, B:C:2\} \) and \( \{B:D:4, E:C\} \) is \( \{A:1, B:C:2, D:4, E:C\} \).

**Definition 3.3** A value is **terminal** if it is (a) the empty map, (b) a base value or (c) a path.

Given a path \( <v_1, \ldots, v_n> \) and a terminal \( t \) we can construct a value \( \{v_1 : \{\ldots\{v_n : t\}\ldots\} \} \) in which each (top level) map has a single-element domain. Any value can be specified by giving its terminals together with their associated identifiers:

**Prop 3.4** Any value can be expressed as the deep union of a set \( M \) of path/terminal pairs where the set of paths \( \{p \mid (p, t) \in M\} \) is independent.

(A set of paths is independent if no one is a prefix of another) We call this structure the **path expansion** of a value.

Example: The following set of path/terminal pairs constructs the first two “tuples” in the database in fig. 1

\(<\text{EEmps}.\{\text{Id}:11\}.\text{Id}> 11 \\
<\text{EEmps}.\{\text{Id}:11\}.\text{Name}> "\text{Kim}" \\
<\text{EEmps}.\{\text{Id}:11\}.\text{Rate}> 50 \\
<\text{EEmps}.\{\text{Id}:22\}.\text{Id}> 22 \\
<\text{EEmps}.\{\text{Id}:22\}.\text{Name}> "\text{Bob}" \\
<\text{EEmps}.\{\text{Id}:22\}.\text{Rate}> 75 \\
<\text{EEmps}.\{\text{Id}:22\}.\text{Manager}> <\text{EEmps}.\{\text{Id}:11\}>

**Prop 3.5** If \( v_1 \) and \( v_2 \) are consistent values, so is deepu\( (v_1, v_2) \).

An operation similar to deep union is **deep update**. It is obtained by replacing the definition of \( g(y_1, y_2) \) in the definition of deep union by:

- **deepu** \( (y_1, y_2) \) if \( y_1 \) and \( y_2 \) are both maps,
- \( y_2 \) otherwise.

Example: A deep update of the database in figure 1 with \( \{\text{EEmps} : \{\text{Id:11} : \{\text{Age:35, Rate:71}\}\}\} \) will insert the pair \( \text{Age:35} \) and modify the rate to 71 in the first “tuple”.

Variations on deep union and deep update are used in file synchronization systems [3]. AcceDB is interesting, because AcceDB values are best modeled by values in which every terminal is the empty map. We shall call such values **null terminated**. They can also be used to describe complex object databases in which every attribute is set-valued. On null-terminated values, deep update and deep union coincide. AcceDB has some additional machinery that allows a map to be modified, and this could be adopted for more general setting we are using here.

**Application views.** The application \( v(v') \) of one value to another does not, in general, yield a consistent value. However we can obtain a consistent application by taking its path expansion and

- removing any path/terminal pairs that contain a path for which \( v' \) is not a prefix and then
- removing \( v' \) from the head of those paths that remain.

For example, the application view of the database in figure 1 is

\[
\{ \{\text{Id:11}\} : \{\text{Name: "Kim", Rate: 50}\}, \\
\{\text{Id:22}\} : \{\text{Name: "Bob", Rate: 75, Manager: \langle\{\text{Id:11}\}\rangle}\} \\
\{\text{Id:32}\} : \{\text{Name: "Smith", Rate: 55, Manager: \langle\{\text{Id:11}\}\rangle}\} \}
\]

This is one simple operation that can be used to construct a consistent view. Another interesting method is through the use of types.

### 4 Types

There are various approaches to setting up type systems to describe our deterministic data model. We shall adopt one that is close to conventional object-oriented type systems. For this it is useful to have two different map types. Homogeneous maps such as int\(\times\)string for arrays and heterogeneous maps which are used to represent tuples. To this end we introduce a syntactic class of tuple labels and type maps from labels to values in the same way that one types tuple types. Our syntax for types is then

\[
\tau ::= b_1 | \ldots | b_n | \tau\times\tau | \{l: \tau, \ldots, l: \tau\} | \tau, \ldots, \tau
\]

Where the \( b_i \) are the base types and \( l \) ranges over labels. Apart from the use of a map type rather than a set type, the main difference between this and object-oriented[9] or complex-object[13] type systems is the addition of an explicit type for paths. For example, here is a type that might be used to characterize the database in figure 1...
5 A Query Language

Query languages for semi-structured data (and some languages for structured data [7]) [2, 5, 8] have a simple generic formulation:

\[
\begin{align*}
\text{collect} & \quad e \\
\text{where} & \quad p_1 \leftarrow e_1, \quad \ldots \\
& \quad p_n \leftarrow e_n \\
\text{condition} & \end{align*}
\]

In this, the \( p_i \) are patterns that introduce variables, \( \text{condition} \) is a predicate on those variables and \( e \) is an expression in some language which may involve the variables introduced by the \( p_i \). The meaning of such an expression is to find all matches of the patterns \( p_i \) to the structures specified by \( e_i \), and for each set of variable bindings that satisfy \( \text{condition} \) evaluate \( e \). The result is the set of all generated values.

In our case we shall want to generate a map rather than a set. Therefore we take the meaning of a query to be the \textit{deep union} of all the collected values of \( e \).

For the time being we shall take the syntax of both patterns and expressions to be our previous syntax of data augmented with variables:

\[
\begin{align*}
& c := b_1 \mid \ldots \mid b_n \mid \{e_1, \ldots, e_m : e_m\} \mid \langle e_1, \ldots, e_m \rangle \mid x
\end{align*}
\]

Here \( c_i \) ranges over the constants of base type \( b_i \), and \( x \) ranges over variables, which will be represented as lowercase single letters in our following examples.

We define structure matching semantically as follows:

\textbf{Definition 5.1} \( w \) is a \textit{substructure} of \( v \) if

- \( w \) and \( v \) are both constants or paths and \( w = v \), or
- \( w \) and \( v \) are both maps and for each pair \((x,y)\) in \( w \) there is a pair \((x,z)\) in \( v \) such that \( y \) is a substructure of \( z \)

It is a consequence of our model that \( w \) can occur in “at most one way” as a substructure of \( v \). Due to the similar syntax, the notion of substructure extends naturally to expressions and patterns when treating variables as constants.

A pattern \( p \) \textit{matches} a structure \( v \) if there is an assignment of structures to the variables in \( p \) that makes it a substructure of \( v \).

For example,

\[
\begin{align*}
\text{collect} & \quad \{\{\text{Id}: i\} : \{\text{Name}: n\}\} \\
\text{where} & \quad \{\text{Emps} : \{\{\text{Name}: n\} : \{\text{Rate} : r\}\} \leftarrow DB, \\
& \quad r > 50
\end{align*}
\]

applied to the database in figure 1, yields the set

\[
\begin{align*}
\{\text{Id: 22}\} : \{\text{Name} : ‘Bob’\}, \\
\{\text{Id: 32}\} : \{\text{Name} : ‘Smith’\}
\end{align*}
\]

Another example is \text{collect} \( \{y : \{x : z\}\} \) where \( \{x : \{y : z\}\} \leftarrow DB \). This is an unnest/nest operation. When
applied to a database of type $\text{int} \times \text{int} \times \tau$, it expresses matrix transposition.

**Elementary Queries.** Towards a normal form for our queries, we introduce the class of elementary queries as the queries of general form:

$$\text{collect} \ \text{pathPattern} \ \text{where} \ \text{pathPattern}_1 \leftarrow v_1,$$

$$\ldots$$

$$\text{pathPattern}_n \leftarrow v_n \ \text{condition}$$

Here, each of $\text{PathPattern}$ and $\text{PathPattern}_i$ are obtained from a path, just like values are obtained from paths in the path expansion. Each $v_i$ is a value (a database or a constant). While the second of our above examples is an elementary query, the first one is not.

**Prop 5.2 (Query Decomposition)** Any query in general form can be expressed as a deep union of elementary queries.

**Query Composition.** We address the following question: given two queries in our $\text{collect} \ \ldots \ \text{where} \ \ldots$ syntax, can we express their composition, and if so how? From a semantic point of view, this composition obviously exists due to the fact that our queries run over maps and produce maps. Notice however that our expression syntax rules out nested queries within the where clause.

The following proposition holds nonetheless:

**Prop 5.3 (Query Composition)** Let $Q_1(T) = \text{collect } e_1$ where $p_1 \leftarrow T$ and $Q_2(S) = \text{collect } e_2$ where $p_2 \leftarrow S$. Then $Q_1 \circ Q_2(S) = \text{deepu}_{1 \leq i \leq 2} (\text{collect } e_1 \theta_i \text{ where } p_2 \theta_i \leftarrow S)$, where $(\theta_i)_{1 \leq i \leq 2}$ are all substitutions such that $\theta_i$ (most generally) unifies $p_1$ with some substructure of $e_2$.

For example, $Q_1 = \text{collect } v$ where $\{A : v\} \leftarrow T$ and $Q_2 = \text{collect } \{y : \{x : z\}\}$ where $\{x : \{y : z\}\} \leftarrow S$ compose to collect $\{x : z\}$ where $\{x : \{A : z\}\} \leftarrow S$. Similarly, $Q_1$ and $Q_2 = \text{collect } \{y : \{x : z\}, u : \{x : w\}\}$ where $\{x : \{y : z, u : w\}\} \leftarrow S$ compose to $\text{deepu} (\text{collect } \{x : z\} \text{ where } \{x : \{A : z, u : w\}\} \leftarrow S, \text{collect } \{x : w\} \text{ where } \{x : \{y : z, A : w\}\} \leftarrow S)$.

**Well-Defined Queries.** Not all queries are well-defined, for example $\text{collect } \{x : y\}$ where $\{y : x\} \leftarrow \{2 : 1, 3 : 1\}$ will cause an error when we attempt to take the deep union of $\{1 : 2\}$ and $\{1 : 3\}$. When can we guarantee that a deep union will exist?

Notice that in the null-terminated model, the answer to this question is trivially "always". The following discussion is restricted to the model with base-type terminated paths, and within this model, to elementary queries.

**Definition 5.4** A variable dependency is a pair of sets $X, Y$ of variables $X \rightarrow Y$. It is used to express the fact that if in some matching, the variables in $X$ have been instantiated, then there will be at most one instantiation for the variables in $Y$.

The variable dependencies generated by pattern (or expression) $e$ is the set

$$\{f_v(p) \rightarrow f_v(t) \mid (p, t) \in \text{the path expansion of } e\},$$

where $f_v(e)$ is the set of variables occurring in $e$. For example, $(x : \{y : \{1 : z, 2 : w\}\})$ generates the dependencies $(x, y) \rightarrow \{z\}$ and $(x, y) \rightarrow \{w\}$. $(x : \{y : z\}, z : x)$ generates the dependencies $(x, y) \rightarrow \{z\}$ and $(z) \rightarrow y$.

**Prop 5.5 (Well-definedness)** A sufficient condition for the elementary query

$$\text{collect } p \ \text{where} \ \ p_1 \leftarrow v_1,$$

$$\ldots$$

$$p_n \leftarrow v_n \ \text{condition}$$

to be well-defined is that the variable dependencies of $p$ can be derived (using Armstrong’s axioms) from the union of the variable dependencies of $p_1 \ldots p_n$.

Conversely, if the variable dependencies of $p$ cannot be so derived, then there are values for $v_1 \ldots v_n$ and for condition (true) for which the query is ill defined.

6 Conclusions

This work on a deterministic model for semi-structured data is still ongoing. One of the issues we are currently working on is the characterization of the class of queries which yields consistent views. (Analogous to application views in section 3.)

Another issue is extending the syntax of expressions to allow for deep applications. This extension will allow expressing queries that navigate through the database by path chasing. For example, recalling the database in figure 1, we obtain the names of employees with over 100 hours per client as

$$\text{collect } \{\{\text{Id : i} : \{\text{Name : n}\}\}$$

where

$$\{\text{Contracts} : \{\{\text{Emp : p, Client : c} : \{\text{Hours : h}\}\} \leftarrow DB, \{\text{Emps} : \{\{\text{Id : i} : \{\text{Name : n}\}\}\} \leftarrow DB(p), h > 100$$

References


