Programming Constructs for Unstructured Data

Peter Buneman* Susan Davidson*
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{peter,susan}@cis.upenn.edu

Dan Suciu†
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA
suciu@research.att.com

Abstract

We investigate languages for querying and transforming *unstructured* data by which we mean languages than can be used without knowledge of the structure (schema) of the database. There are two reasons for wanting to do this. First, some data models have emerged in which the schema is either completely absent or only provides weak constraints on the data. Second, it is sometimes convenient, for the purposes of browsing, to query the database without reference to the schema. For example one may want to "grep" all character strings in the database, or one might want to find the information associated with a certain field name no matter where it occurs in the database.

This paper introduces a labelled tree model of data and investigates various programming structures for querying and transforming such data. In particular, it considers various restrictions of structural recursion that give rise to well-defined queries even when the input data contains cycles. It also discusses issues of observable equivalence of such structures.

1 Introduction

We investigate languages for querying and transforming *unstructured* data by which we mean languages than can be used without knowledge of the structure (schema) of the database. There are two reasons for wanting to do this. First, some data models have emerged in which the schema is either completely absent or only provides weak constraints on the data. Second, it is sometimes convenient, for the purposes of browsing, to query the database without reference to the schema. For example one may want to "grep" all character strings in the database, or one might want to find the information associated with a certain field name no matter where it occurs in the database.

The idea of using labeled trees for this purpose has been suggested by two data models. ACeDB (A C. elegans Database) [11] is a database system popular with biologists. It has a schema, but this only places very weak constraints on the database since any field in the deeply nested records that are common in ACeDB can be null. Recently Tsimmis [9] has been proposed as a data model for heterogeneous data integration. In Tsimmis there is no schema. The "type" is interpreted by the user from labels in the structure, which is quite flexible. In particular, a Tsimmis structure may be interpreted as a record or as a set. There is an analogy here with the dynamic type system of Lisp, whose one basic data structure, the s-expression, may be used to represent lists, trees, association lists, lambda terms, etc. We go a step further and blur the distinction between set nodes and

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

†Supported in part by NSF Grant CCR-90-57570, ONR Contract NOOO14-93-11284, and by a fellowship from the Institute for Research in Cognitive Science

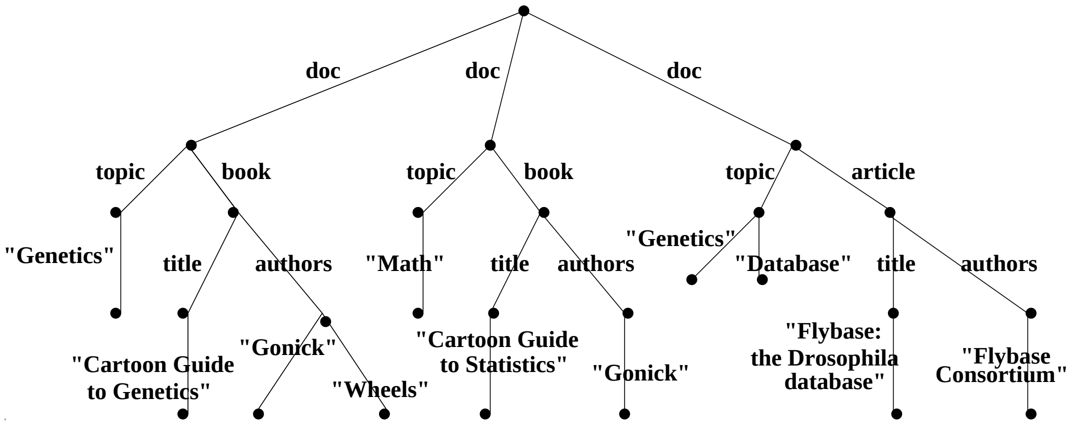

Figure 1: Labeled tree representation of a bibliographic database, *bib*.

record nodes in the labeled trees. We believe that this untyped data model is advantageous for querying databases with unknown or only partially known structure: Christophides et al. [6] consider a query language with similar goals for an object-oriented data model.

The approach we shall take is to extend *structural recursion* to labeled trees. This poses some interesting problems: first, it is no longer "flat" structural recursion, so that the usual syntactic forms and optimizations for collection types such as lists bags and sets may not be relevant. Second, we shall want to examine the possibility that the values we are manipulating may be cyclic. It is common in ACeDB, and generally in object-oriented databases, for objects to refer to each other, allowing the possibility of arbitrarily "deep" queries. Of course, such cyclic structures are usually constructed through the use of a reference/pointer type; however query languages are insensitive to these object identities and perform automatic dereferencing. We therefore want to understand what programs are well defined when we are allowed to make unbounded searches in the database.

This work is preliminary and serves only to describe languages that may be useful for unstructured data. While we believe that there are sound principles for justifying this choice of languages, they are at present mostly "articles of faith". The paper is organized as follows. After specifying the data structure of interest, we first develop a variant of nested relational algebra which gives us the ability to construct queries to a fixed depth. Next we extend the idea of structural recursion to perform queries that can reach data at arbitrary depths in the tree. Finally we examine restrictions of this language that work on cyclic data. By a database "query" we usually understand a program that produces a relatively simple output from a complex input – the database. In what follows we are interested in producing data structures that may be as complex as the input. This is the problem of *transforming* databases, which is of paramount importance in heterogeneous database systems.

2 A labeled tree data type

As is common in this area we shall take a bibliographic database as a running example. The diagram in Figure 1 shows an edge-labeled tree. At the top level we see three edges labeled *doc* indicating a set of documents. The first such document displayed is a tree with two distinct labels *topic* and *book* indicating a record.

The labels on the edges are drawn from some collection of basic types. For the sake of consistency with the systems mentioned above, we shall consider the type *label* to be the discriminated union of a number of basic types: *character strings* such as *"Math", "Wheels"*; *numbers* such as 4, 3.1415; and *symbols* such as *'doc, 'article* using the lisp notation for quoting (the quote mark is not shown on the symbol edges in Figure 1). In general, symbols are used to mark internal edges, and other constants such as strings and numbers are used at the leaves, but this is not demanded by our model.

Having fixed a data type *label*, we can now define the type of a labeled *tree* to be a set of pairs, each consisting of a label and another tree. Using $\mathcal{P}_{fin}(S)$ for the finite subsets of S, we can describe a labeled tree type by the recursive type equation

$$\mathit{tree} = \mathcal{P}_{fin}(\mathit{label} \times \mathit{tree})$$

Before proceeding further we should remark that there are some differences between this type and the models used in Tsimmis and ACeDB. First of all Tsimmis [9] labels nodes instead of edges: in the framework of our query language constructs the two styles of labelings are equivalent. Secondly Tsimmis attaches values of base types such as *num* and *string* to the terminal nodes of the tree, and the edges are labeled only with symbols. Tsimmis also has object identities associated with the internal nodes. The transformation from Tsimmis is straightforward: we represent terminal nodes by terminal edges; and we may introduce object identities by simply adding a new object-identity base type. ACeDB, is much closer to our presentation in that numbers, strings etc. may be attached to non-terminal edges. It also allows one to build cyclic structures, which we shall discuss later. The transformation from ACeDB is obtained essentially by transferring label information from the schema to the data; and a similar technique may be used to represent other databases as trees.

We now describe constructors for the type *tree*. Trees are sets, so we have ϕ for the empty set and $e_1 \cup e_2$ to construct the union of sets e_1 and e_2. In addition we have the expression $\{a \Rightarrow e\}$ to describe a singleton set consisting of a tree formed by attaching the edge labeled with a to the root of the tree e. The types of these constructors are as follows:

$$\begin{array}{rcl} \phi & : & \mathit{tree} \\ \{\_ \Rightarrow \_\} & : & \mathit{label} \times \mathit{tree} \rightarrow \mathit{tree} \\ \_ \cup \_ & : & \mathit{tree} \times \mathit{tree} \rightarrow \mathit{tree} \end{array}$$

We shall also make use of the following abbreviations for constructing trees: $\{a_1 \Rightarrow e_1, a_2 \Rightarrow e_2, \ldots a_n \Rightarrow e_n\}$ for $\{a_1 \Rightarrow e_1\} \cup \{a_2 \Rightarrow e_2\} \cup \ldots \cup \{a_n \Rightarrow e_n\}$. Also $a \Rightarrow \phi$, appearing within $\{\ldots\}$ may be abbreviated to a. Thus $\{1, 2, 3\}$ is an abbreviation for the "flat" tree $\{1 \Rightarrow \phi, 2 \Rightarrow \phi, 3 \Rightarrow \phi\}$

As a more elaborate example, the tree depicted in Figure 1 can be built with the following syntax:

$$\begin{array}{l} \mathit{bib} = \{\, \text{'}\mathit{doc} \Rightarrow \{\, \text{'}\mathit{topic} \Rightarrow \{\text{“}\mathit{Genetics}\text{”}\}, \\ \qquad\qquad\quad \text{'}\mathit{book} \Rightarrow \{\, \text{'}\mathit{title} \Rightarrow \{\text{“}\mathit{Cartoon\ Guide\ to\ Genetics}\text{”}\}, \\ \qquad\qquad\qquad\qquad \text{'}\mathit{authors} \Rightarrow \{\text{“}\mathit{Gonick}\text{”}, \text{“}\mathit{Wheels}\text{”}\}\}\}, \\ \qquad\quad \text{'}\mathit{doc} \Rightarrow \{\, \text{'}\mathit{topic} \Rightarrow \{\text{“}\mathit{Math}\text{”}\}, \\ \qquad\qquad\quad \text{'}\mathit{book} \Rightarrow \{\, \text{'}\mathit{title} \Rightarrow \{\text{“}\mathit{Cartoon\ Guide\ to\ Statistics}\text{”}\}, \\ \qquad\qquad\qquad\qquad \text{'}\mathit{authors} \Rightarrow \{\text{“}\mathit{Gonick}\text{”}\}\}\}, \\ \qquad\quad \text{'}\mathit{doc} \Rightarrow \{\, \text{'}\mathit{topic} \Rightarrow \{\text{“}\mathit{Genetics}\text{”}, \text{“}\mathit{Database}\text{”}\}, \\ \qquad\qquad\quad \text{'}\mathit{article} \Rightarrow \{\, \text{'}\mathit{title} \Rightarrow \{\text{“}\mathit{FlyBase\text{-}the\ Drosophila\ database}\text{”}\}, \\ \qquad\qquad\qquad\qquad\quad \text{'}\mathit{authors} \Rightarrow \{\text{“}\mathit{The\ FlyBase\ Consortium}\text{”}\}\}\}\}\} \end{array}$$

3 Nested relational algebra on trees

The previous section gave a syntax for constructing trees. We now extend this to the syntax of a programming language for trees. To our types *label* and *tree* we add a boolean type *bool* with the usual boolean connectives[1].

We also add

- An equality test $a = b$ on labels. Equality is of type $\mathit{label} \times \mathit{label} \rightarrow \mathit{bool}$.
- An emptiness test $\mathit{null}(t)$ on trees. $\mathit{null} : \mathit{tree} \rightarrow \mathit{bool}$
- A conditional $\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2$ in which b is a boolean expression and e_1, e_2 denote trees.

Since trees are necessarily sets, we first consider structural recursion on sets as the basic programming paradigm, and following [3] use the restricted form of structural recursion given by functions h of the form

$$\begin{array}{lcl} h(\phi) & = & \phi \\ h(\{a \Rightarrow t\}) & = & f(a,t) \\ h(t_1 \cup t_2) & = & h(t_1) \cup h(t_2) \end{array}$$

In this, the meaning of the function h of type $\mathit{tree} \rightarrow \mathit{tree}$ is determined by the function $f : \mathit{label} \times \mathit{tree} \rightarrow \mathit{tree}$. Note that this is a mathematical definition, which suggests an implementation. The syntax we will use for $h(S)$ when the function f is defined by $f(a,t) = e$ is $\mathit{ext}(\lambda(a,t).e)(S)$.

An example of this form of definition is a selection function:

$$\begin{array}{lcl} \mathit{sel}(b)(\phi) & = & \phi \\ \mathit{sel}(b)(\{a \Rightarrow t\}) & = & \mathit{if}\ a = b\ \mathit{then}\ \{a \Rightarrow t\}\ \mathit{else}\ \phi \\ \mathit{sel}(b)(t_1 \cup t_2) & = & \mathit{sel}(b)(t_1) \cup \mathit{sel}(b)(t_2) \end{array}$$

In our syntax, $\mathit{sel}(b)(S)$ can be written as $\mathit{ext}(\lambda(a,t).\mathit{if}\ a = b\ \mathit{then}\ \{a \Rightarrow t\}\ \mathit{else}\ \phi)(S)$. Its effect is to discard from S all edges that are not labeled with b, together with their subtrees. For example, $\mathit{sel}(1)(\{1 \Rightarrow \{10\}, 2 \Rightarrow \{20\}\}) = \{1 \Rightarrow \{10\}\}$.

Another useful function is projection, defined as $\mathit{proj}(b)(S) \stackrel{\mathrm{def}}{=} \mathit{ext}(\lambda(a,t).\mathit{if}\ a = b\ \mathit{then}\ t\ \mathit{else}\ \phi)(S)$. This function takes the union of the trees at the ends of b edges and discards the others. Note how it differs from $\mathit{sel}(b)$: $\mathit{proj}(1)(\{1 \Rightarrow \{10\}, 2 \Rightarrow \{20\}\}) = \{10\}$.

A flattening function defined as $\mathit{flat}(S) \stackrel{\mathrm{def}}{=} \mathit{ext}(\lambda(a,t).t)(S)$ will also be useful. This function removes one level of edges out of the root and takes the union of the subtrees at their ends: $\mathit{flat}(\{1 \Rightarrow \{10\}, 2 \Rightarrow \{20\}\}) = \{10, 20\}$.

To summarize the language at this point, we assume we have an infinite collection of typed variables for labels (ranged over by a) and for trees (ranged over by t). In addition we have a set of constants of type *label* as described above. An expression in the language is built up from the variables and constants with the following rules:

$$\frac{}{\phi : \mathit{tree}} \qquad \frac{l : \mathit{label} \quad e : \mathit{tree}}{\{l \Rightarrow e\} : \mathit{tree}} \qquad \frac{e_1 : \mathit{tree} \quad e_2 : \mathit{tree}}{e_1 \cup e_2 : \mathit{tree}} \qquad \frac{e : \mathit{tree} \quad e' : \mathit{tree}}{\mathit{ext}(\lambda(a,t).e')e : \mathit{tree}}$$

$$\frac{e : \mathit{tree}}{\mathit{null}\ e : \mathit{bool}} \qquad \frac{l_1 : \mathit{label} \quad l_2 : \mathit{label}}{l_1 = l_2 : \mathit{bool}} \qquad \frac{b : \mathit{bool} \quad e_1 : \mathit{tree} \quad e_2 : \mathit{tree}}{\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \mathit{tree}}$$

[1] The introduction of a boolean type is inessential. We can simulate *false* with the empty set and *true* with some non-empty set. See [3] for details.

We also assume the boolean constants and operations with the obvious typing rules together with other appropriate operations on labels. We shall call this language EXT.

Nested Relational Algebra. We now have a language equivalent in expressive power to the nested relational algebra, since it includes all the operations described in [3]. Note that although we have not introduced an explicit pairing operation into EXT, it can be simulated with the operators already available. If we fix two labels, $1, 2$, the pair e_1, e_2 can be expressed as $\{1 \Rightarrow e_1, 2 \Rightarrow e_2\}$ and the projection operations are $proj(1), proj(2)$. In particular, we may simulate a flat relation by constructing for each tuple $(v_1, \ldots, v_n)$ in relation $R(A_1, \ldots A_n)$ a tree $\{\,'R \Rightarrow \{\,'A_1 \Rightarrow \{v_1\} \ldots\, 'A_n \Rightarrow \{v_n\}\}\}$ and taking the union of all such trees.

To illustrate the types of queries and transformations that we can perform with EXT we give some examples in the spirit of [9]. To simplify their presentation, we will use the following abbreviations: $e.a$ for $proj(a)(e)$, $e{\uparrow}a$ for $sel(a)(e)$, and $a\ in\ e$ for $\neg null(sel(a)(e))$.

Example 1: Find the titles of all books on Genetics.
$ext(\lambda(a,t).if\ (a = \,'doc)\ and\ (\text{“Genetics”}\ in\ t.\,'topic)\ then\ t.\,'book{\uparrow}\,'title\ else\ \phi)(bib)$

Example 2: Find the authors of all documents, regardless of the type of document.
$ext(\lambda(a,t).if\ a = \,'doc\ then\ (flat\ t){\uparrow}\,'authors\ else\ \phi)(bib)$

Example 3: Find the title and topic of all books by Gonick and Wheels.
$ext(\lambda(a,t).if\ (a = \,'doc)\ and\ (\text{“Gonick”}\ in\ t.\,'book.\,'author)\ and\ (\text{“Wheels”}\ in\ t.\,'book.\,'author)$
$then\ \{\,'book \Rightarrow \{t{\uparrow}\,'topic\} \cup \{t.\,'book{\uparrow}\,'title\}\}\ else\ \phi)(bib)$

The last example does not return a subtree of the original tree, and illustrates how the result can restructure information. Such restructuring cannot be performed in [9].

It should also be observed that the queries in these examples assume a particular structure on the trees, i.e. that the labels of interest appear at predetermined depths. In the next section, we will see how to specify queries which operate on trees in which labels can appear at arbitrary depths.

4 Structural recursion on trees

We now consider a form of structural recursion that one would naturally associate with trees.

$$
\begin{array}{lcl}
h(\phi) & = & \phi \\
h(\{a \Rightarrow t\}) & = & f(a, h(t)) \\
h(t_1 \cup t_2) & = & h(t_1) \cup h(t_2)
\end{array}
$$

The only difference between this and our previous form of structural recursion is that h acts recursively on the subtrees of a tree. As before we will use the syntax $text(\lambda(a,r).e)(S)$ for $h(S)$ when the function f is defined by $f(a,r) = e$. Intuitively the parameter r in f stands for the recursive call of $h(t)$.

For example, to change each $'topic$ label to a $'subject$ label we may use the function $change\_lab$ defined by

$$
\begin{array}{lcl}
change\_lab(\phi) & = & \phi \\
change\_lab(\{a \Rightarrow t\}) & = & if\ a = \,'topic\ then\ \{\,'subject \Rightarrow change\_lab(t)\}\ else\ \{a \Rightarrow change\_lab(t)\} \\
change\_lab(t_1 \cup t_2) & = & change\_lab(t_1) \cup change\_lab(t_2)
\end{array}
$$

This will change labels at any depth in the tree. It is expressed using $text(\_)$ as

$$change\_lab(S) \stackrel{\text{def}}{=} text(\lambda(a,r).if\ a = {'topic}\ then\ \{{'subject}\Rightarrow r\}\ else\ \{a\Rightarrow r\})(S)$$

We may also write a selection function that operates over the whole tree. $tsel(p)(S)$ selects only those edges in S that satisfy p; the other edges are lost and their subtrees become inaccessible. It is defined by

$$tsel(p)(S) \stackrel{\text{def}}{=} text(\lambda(a,r).if\ p(a)\ then\ \{a\Rightarrow r\}\ else\ \phi)(S)$$

Applying this to the *bib* structure with the predicate $p(x) = \neg(x={'topic})$ will result in the topic labels and the associated strings being removed from the tree.

We may also build a "flat" tree of all the edges in in a tree with

$$flat\_trees(S) \stackrel{\text{def}}{=} text(\lambda(a,r).\{a\} \cup r)(S)$$

Then $flat\_trees(bib)$ results in $\{{'doc},\ {'topic},\ \text{“Genetics”},\ {'book},\ \ldots\}$. With such a transformation and the use of the discriminating function for strings, we can easily find all the strings in the database.

A more interesting example is to find a tree containing the set of all paths from the root of the tree. We represent a path by a list, or "vertical" tree, so that the path consisting of the sequence of labels *'doc*, *'book*, *'title* is $\{{'doc}\Rightarrow\{{'book}\Rightarrow\{{'title}\}\}\}$. We can obtain the set of all paths with

$$all\_paths(S) \stackrel{\text{def}}{=} text(\lambda(a,r).\{a\} \cup ext(\lambda(a_1,r_1).\{a\Rightarrow\{a_1\Rightarrow r_1\}\})(r))(S)$$

In this expression, r is bound recursively to all the paths of the subtree below the edge a. The set of paths we want includes the single edge a together with the the paths that are formed by tacking a onto the beginning of each of the paths in r, which is done with an application of $ext(\_)$. The result of this query will be

$$
\begin{array}{l}
\{{'doc},\\
\quad {'doc}\Rightarrow\{{'topic}\},\\
\quad {'doc}\Rightarrow\{{'topic}\Rightarrow\{\text{"Genetics"}\}\},\\
\quad {'doc}\Rightarrow\{{'book}\},\\
\quad {'doc}\Rightarrow\{{'book}\Rightarrow\{{'title}\}\},\\
\quad \ldots\\
\}
\end{array}
$$

As a final example of the use of $text(\_)$, consider the expression

$$blow\_up(S) \stackrel{\text{def}}{=} text(\lambda(a,r).\{a\Rightarrow r\}\ \cup\ r)(S)$$

When S the linear tree $\{1\Rightarrow\{2\Rightarrow\{3\}\}\}$ $blow\_up(S)$ produces $\{1\Rightarrow\{2\Rightarrow\{3\},3\},2\Rightarrow\{3\},3\}$. When S is a linear tree with n edges it produces a tree with $2^n - 1$ edges. We shall see later that this apparent growth does not imply that we are performing transformations that are outside PTIME.

The preceding examples show that a number of queries that are expressed with "path variables" [9] can be computed in EXT together with $text(\_)$. We will refer to this extended version of EXT as $TEXT$. However there is one extremely useful query which presents some difficulty: it is that of computing the union of a tree with all of its subtrees. A refinement of this problem is to obtain all the edges that satisfy a certain property together with their subtrees. For example, we might want all the books in the *bib* structure. In order to build such a tree we need a more general form of recursion

$$
\begin{array}{lcl}
h(\phi) & = & \phi \\
h(\{a \Rightarrow t\}) & = & f(a, t, h(t)) \\
h(t_1 \cup t_2) & = & h(t_1) \cup h(t_2)
\end{array}
$$

in which the function f is now a three-place function taking the edge label a, the input subtree t, and the result of recursively computing h on that subtree. We use the form *text'*$(\lambda(a,t,r).e)(S)$ for this form, where the function f is defined by $f(a,t,r) = e$. The union of a tree S with all of its subtrees is now given by:

$$
\mathit{all\_trees}(S) \stackrel{\text{def}}{=} \mathit{text'}(\lambda(a,t,r).\{a \Rightarrow t\} \cup r))(S)
$$

Note how *all_trees* differs from *blow_up*: $\mathit{all\_trees}(\{1 \Rightarrow \{2 \Rightarrow \{3\}\}\}) = \{1 \Rightarrow \{2 \Rightarrow \{3\}\}, 2 \Rightarrow \{3\}, \{3\}\}$.

As a further example, to find all the books in the *bib* database together with their subtrees we can write

$$
\mathit{text'}(\lambda(a,t,r).(\mathit{if}\ a = \text{'}\mathit{book}\ \mathit{then}\ \{a \Rightarrow t\}\ \mathit{else}\ \phi) \cup r))(\mathit{bib})
$$

Note that if we have one *'book* tree below (or inside) another, this function will extract both of them.

It is possible to implement *text'*(_) in *TEXT*, so that this form of structural recursion presents nothing new. However, the implementation involves the use of projection, which can cause problems when we consider operations on cyclic structures. The direct use *text'*(_) is more closely related to a form of structural recursion that we shall now examine.

5 Cyclic structures

The languages *EXT* and *TEXT* operate on labeled trees. Surprisingly, *EXT* and an important fragment of *TEXT*, which we call *VEXT*, can be extended naturally from trees to cyclic structures. This is due to the fact that the queries in *EXT* and *VEXT* can be computed by independently processing each edge of a cyclic structure, without needing to chase every path in the structure.

Syntactically, we describe cyclic structures with the aid of *variables* and *equations* defining these variables. Semantically, cyclic structures are rooted, labeled graphs.

Consider the syntactic specification of a cyclic structure given in Figure 2. It uses the variables X_1, X_2, Y_1, Y_2, and four equations defining them. We recover a tree for such a specification by textually substituting in $X_1 \cup X_2$ each variable with the right hand side of the equation defining it, and by repeating this process until all variables are eliminated, thus *unfolding* a labeled tree. E.g. consider the same example from Figure 2, but with the definitions of Y_1, Y_2 changed to: $Y_1 =$ *"Gonick"* and $Y_2 =$ *"Wheels"*. Then by unfolding we get a subtree of the tree of Figure 1. In general the unfolded tree may be infinite. Note that we may have different syntactic specifications denoting the same tree: these should be regarded as equal. Also, exactly those (infinite) labeled trees are meanings of syntactic specifications of cyclic structures, which are *rational*, i.e. for which the set of subtrees is finite.

Formally, a syntactic specification of a cyclic structure is:

$$
\begin{array}{l}
e\ \mathit{where} \\
\quad X_1 = e_1 \\
\quad \ldots
\end{array}
$$

$$
\begin{array}{l}
X_1 \cup X_2 \ \textit{where} \\
\qquad \begin{array}{rcl}
X_1 & = & \{doc \Rightarrow \{topic \Rightarrow \{\text{"}Genetics\text{"}\}, \\
& & \qquad book \Rightarrow \{title \Rightarrow \text{"}Cartoon\ Guide\ to\ Genetics\text{"}, \\
& & \qquad authors \Rightarrow Y_1 \cup Y_2\ \}\}\} \\
X_2 & = & \{doc \Rightarrow \{topic \Rightarrow \{\text{"}Math\text{"}\}, \\
& & \qquad book \Rightarrow \{title \Rightarrow \{\text{"}Cartoon\ Guide\ to\ Statistics\text{"}\}, \\
& & \qquad authors \Rightarrow Y_1\ \}\}\} \\
Y_1 & = & \{\text{"}Gonick\text{"}, \\
& & \ papers \Rightarrow X_1 \cup X_2\} \\
Y_2 & = & \{\text{"}Wheels\text{"}, \\
& & \ papers \Rightarrow X_1\}
\end{array}
\end{array}
$$

Figure 2: A specification of a cyclic structure

$$X_k \quad = \quad e_k$$

Here $e, e_1, \ldots, e_k$ are *labeled trees with markers* $X_1, \ldots, X_k$. More precisely, they are expressions built up using the three constructors $\emptyset$, $\{\_\Rightarrow\_\}$, and $\cup$, and which may have the variables $X_1, \ldots, X_k$ on their leaves. The type $tree_{X_1,\ldots,X_k}$ of labeled trees with markers $X_1, \ldots, X_k$ is defined by:

$$tree_{X_1,\ldots,X_k} = \mathcal{P}_{fin}(label \times tree_{X_1,\ldots,X_k} \cup \{X_1, \ldots, X_k\})$$

We write $\{\, \text{'}name \Rightarrow \text{"}Joe\ Doe\text{"}\} \cup X_1 \cup X_2$ instead of the official $\{\, \text{'}name \Rightarrow \text{"}Joe\ Doe\text{"}, X_1, X_2\}$.

The semantics of cyclic structures is given by *rooted, labeled graphs*, $G = (V, E, r, l)$. Each such graph has a distinguished vertex $r \in V$ called *the root*, and the edges are labeled with elements from $label \cup \{\varepsilon\}$, i.e. $l : E \to label \cup \{\varepsilon\}$, where ε is a special symbol not occurring in *label*. E.g. the cyclic structure of Figure 2 will be interpreted as the graph given in Figure 3. Notice how we use ε-edges to connect an occurrence of some variable X_i with its definition.

It is on these graphs that we can now define equality. Namely we say that two graphs $G = (V, E, r, l)$, $G' = (V', E', r', l')$ are *bisimilar* iff there exists a binary relation $\sim \subseteq V \times V'$ s.t. (1) $r \sim r'$, and (2) if $v \sim v'$ then for any label $a \in label$, there exists a path $v \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon}\xrightarrow{a} w$ in G iff there exists some path $v' \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon}\xrightarrow{a} w'$ and $w \sim w'$. We state here without proof that two graphs are bisimilar iff their (potentially infinite) unfolded[2] labeled trees are equal; note that according to our definition of a tree (Section 2) the children of a node form a set, i.e. duplicates are eliminated. What we consider as a value is an equivalence class under the bisimilarity relation. Considering duplicate elimination in the context of infinite trees is a departure from the traditional view of rational trees [7]. That is the trees $\{a \Rightarrow \{\}, a \Rightarrow \{\}\}$ and $\{a \Rightarrow \{\}\}$ are equal in our data model, although in the first tree the root has two children, but only one in the second. In the context of infinite trees, bisimulation seems to be the right tool for duplicate elimination. Abiteboul and Kanellakis [1] discuss "pure values", which are also infinite, rational trees, intended for an object-oriented data model. Some of the nodes in these trees can be finite sets too, but duplicate elimination is not discussed.

This gives us an effective procedure for deciding whether two syntactic specifications are equal. Namely (1) convert the two specifications to rooted graphs, and (2) test whether the two graphs are bisimilar. Note that testing for bisimilarity is a PTIME problem. By contrast, testing for graph isomorphism is believed to be outside of PTIME. See [8] for a discussion of the relevance of bisimulation in query languages with object identities.

[2] During the unfolding of a graph, the ε edges will be removed.

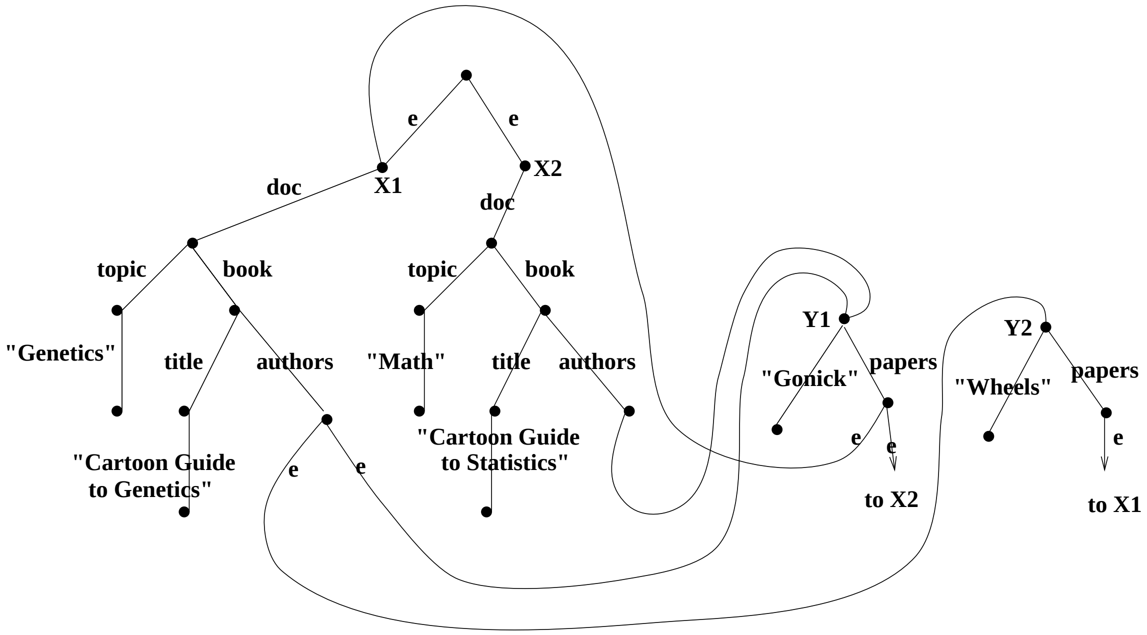

Figure 3: The graph associated to a cyclic structure (the label e stands for ε)

Now we will extend our languages to cyclic structures. First notice that all operations in EXT can be extended straightforwardly to cyclic structures. This is obvious in the case of $\emptyset, \{\_\Rightarrow\_\}$, and $\cup$. To apply *null* of $ext(f)$ to some syntactic specification t, we first have to expose the topmost set in t. For this we convert t into a rooted, labeled graph G, and then restructure into a graph G' bisimilar to G, in which no ε-edge leaves the root (this is always possible). Next we convert G' back to a syntactic specification t', and on t' we apply *null*, or $ext(f)$.

But *text* cannot be extended to cyclic structures. Indeed, consider the query *all_paths*, of Section 4. On some infinite rational tree as input, i.e. a cyclic structure, it will return as output an infinite non-rational tree, which lies outside of our data model.

Fortunately there exists a natural restriction of *text* pointed to us by Val Tannen [10], which allows us to define most of the queries of Section 4, and which generalizes naturally to cyclic structures. We call this restriction *vext*. To define *vext*, we start by discussing another primitive operation, *substitution*. Let X be some variable. We add a new primitive to the ones already mentioned, namely the substitution $subst_X : tree_X \times tree_X \to tree_X$. $subst_X(s,t)$ will simply replace every occurrence of X in s with t. Formally:

$$\begin{aligned}
subst_X(X,t) &\stackrel{\text{def}}{=} t \\
subst_X(\{a\Rightarrow s\},t) &\stackrel{\text{def}}{=} \{a\Rightarrow subst_X(s,t)\} \\
subst_X(t_1 \cup t_2,t) &\stackrel{\text{def}}{=} subst_X(t_1,t) \cup subst_X(t_2,t)
\end{aligned}$$

An intuitive way to think about $subst_X$ is to view it as a generalization of the append operation on lists. For that, let's represent lists as linear trees: $[a_1, a_2, \ldots, a_n]$ becomes $\{a_1\Rightarrow\{a_2\Rightarrow\{\ldots\Rightarrow\{a_n\Rightarrow X\}\}\}\}$. Note that lists end in the marker X, a.k.o. *nil*. If s is the list $[a_1, \ldots, a_n]$ and t is the list $[b_1, \ldots, b_m]$, then $subst_X(s,t)$ is the list $[a_1, \ldots, a_n, b_1, \ldots, b_m]$. Note that had we ended the list s in the empty tree instead of the marker X, i.e. $\{a_1\Rightarrow\{a_2\Rightarrow\{\ldots\Rightarrow\{a_n\Rightarrow\{\}\}\}\}\}$, then $subst_X(s,t)$ would return s. More generaly, a tree s can be thought of as a branching list, having more than one endpoint. Then $subst_X(s,t)$ generalizes append in that it adds t in possible many endpoints of s, namely in all those places where s ends in the marker X.

Finally we are ready to introduce the *vext* construct. Namely for any function $f_X : label \rightarrow tree_X$, $h = vext_X(f_X)$ is defined by:

$$\begin{aligned} h(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\ h(\{a \Rightarrow t\}) &\stackrel{\text{def}}{=} subst_X(f_X(a), h(t)) \\ h(t_1 \cup t_2) &\stackrel{\text{def}}{=} h(t_1) \cup h(t_2) \end{aligned}$$

Again, notice that on "lists" $vext_X$ is simply *ext* on lists.

We now informally describe now how $vext_X(f)$ acts on some syntactic specification of a cyclic structure $t = (e \ where \ X_1 = e_1, \ldots, X_k = e_k)$. The syntactic specification $t' = vext_X(f_X)$ is obtained from t by processing every subexpression of the form $\{a \Rightarrow e'\}$ of $e, e_1, \ldots, e_k$ as follows: we first fetch a fresh variable Y, next replace the subexpression $\{a \Rightarrow e'\}$ with $f_Y(a)$, and finally add the equation $Y = e'$. Intuitively there is a high degree of parallelism in the computation of $vext_X$, which can be visualized even better on rooted, labeled graphs. Namely here, $vext_X(f_X)(S)$ is computed by independently replacing each edge $v \xrightarrow{a} w$ with the tree $f_X(a)$ having v as the root, and by drawing ε-edges from the leaves X of $f_X(a)$ to w; the ε-edges in the original graph are left untouched.

We call *VEXT* the language obtained by extending *EXT* with the *vext* construct. Obviously the following relationships between languages holds:

$$\underbrace{EXT \subseteq VEXT}_{\text{trees and cyclic structures}} \quad \subseteq \quad \underbrace{TEXT}_{\text{trees only}}$$

Here are some of the queries from Section 4 expressed with $vext_X$:

$$\begin{aligned} change\_lab(S) &\stackrel{\text{def}}{=} vext_X(\lambda a.if \ a =' topic \ then \ \{'subject \Rightarrow X\} \ else \ \{a \Rightarrow X\})(S) \\ tsel(p)(S) &\stackrel{\text{def}}{=} vext_X(\lambda a.if \ p(a) \ then \ \{a \Rightarrow X\} \ else \ \emptyset)(S) \\ flat\_trees(S) &\stackrel{\text{def}}{=} vext_X(\lambda a.\{a\} \cup X)(S) \\ blow\_up(S) &\stackrel{\text{def}}{=} vext_X(\lambda a.\{a \Rightarrow X\} \cup X)(S) \end{aligned}$$

Notice how *sharing* allows us to avoid combinatorial explosion of the data structure resulting from the function *blow_up*. E.g. on the input tree $S = \{a_1 \Rightarrow \{a_2 \Rightarrow \{\ldots \{a_n \Rightarrow \emptyset\} \ldots\}\}\}$, $blow\_up(S)$ will return:

$$\begin{aligned} &X_0 \ where \\ &\quad X_0 = \{a_1 \Rightarrow X_1\} \cup X_1 \\ &\quad X_1 = \{a_2 \Rightarrow X_2\} \cup X_2 \\ &\quad \ldots \\ &\quad X_{n-1} = \{a_{n-1} \Rightarrow X_n\} \cup X_n \\ &\quad X_n = \{a_n \Rightarrow \emptyset\} \end{aligned}$$

When we unfold this compact syntactic specification, we obtain a tree with 2^n nodes. In general, we have:

Proposition 5.1 *VEXT is in PTIME.*

Unlike *text* however, $vext_X$ does not seem to be able to express *all_trees*. However we can express *all_trees* with a slight generalization of $vext_X$, from one variable X to an arbitrary number of variables. First we generalize $subst_X$ by defining $subst_{X_1,\ldots,X_n}(t, t_1, \ldots, t_n)$ to be the simultaneous substitution of X_1 with t_1, ..., X_n with t_n in t. Then $vext_{X_1,\ldots,X_n}(f_1, \ldots, f_n)$ is a construct which allows us to define simultaneously n functions $h_1, \ldots, h_n$ by iteration on the cyclic structure (we omit the subscripts $X_1, \ldots, X_n$):

$$\begin{array}{rcl@{\quad}c@{\quad}rcl}
h_1(\emptyset) & \stackrel{\text{def}}{=} & \emptyset & \ldots & h_n(\emptyset) & \stackrel{\text{def}}{=} & \emptyset \\
h_1(\{a \Rightarrow s\}) & \stackrel{\text{def}}{=} & subst(f_1(a), h_1(s), \ldots, h_n(s)) & \ldots & h_n(\{a \Rightarrow s\}) & \stackrel{\text{def}}{=} & subst(f_n(a), h_1(s), \ldots, h_n(s)) \\
h_1(t_1 \cup t_2) & \stackrel{\text{def}}{=} & h_1(t_1) \cup h_1(t_2) & \ldots & h_n(t_1 \cup t_2) & \stackrel{\text{def}}{=} & h_n(t_1) \cup h_n(t_2)
\end{array}$$

Then we can compute *all_path* using $vext_{X_1,X_2}$. More interestingly, for any regular expression on labels, R, we can write in *VEXT* an expression $project_R(S)$ which, on a given tree S, returns the set of all subtrees which can be reached from the root of S using a path in R. E.g. when $R = (a(bc)^*)^*$, then $select_R(S)$ will return the set of all subtrees in S which can be reached from the root by a path of the form $abc \ldots bcabc \ldots bc \ldots abc \ldots bc$. We invite the reader to check that, for any regular expression R, $select_R(S)$ can be written using $vext_{X_1,\ldots,X_n}$. It suffices to take n as the number of states in the deterministic automaton accepting R.

6 Conclusions

We believe that the forms of structural recursion on trees described in this paper offer good prospects for the development of powerful query languages for unstructured data. However to demonstrate this, considerable additional work is needed. First, we need to substantiate the claim that the graph/bisimulation model provides an effective semantics for the language developed in Section 5. Second, there appear to be some optimization techniques for these languages: we discuss some optimization techniques in a related work [5]. An interesting question is how schema information, when it is present, may be used in optimization as suggested in [2]. Third, there are good syntactic forms for languages similar to *EXT* [4] that resemble the syntax of popular database query languages: we address this topic, as well as more powerful language constructs in [5].

Acknowledgment We would like to thank Anthony Kosky and Scott Weinstein for their helpful comments.

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[2] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proceedings of 19th International Conference on Very Large Databases*, pages 73–84, Dublin, Ireland, 1993.

[3] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[5] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996. To appear.

[6] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In Richard Snodgrass and Marianne Winslett, editors, *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[7] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[8] Anthony Kosky. Observational properties of databases with object identity. Technical Report MS-CIS-95-20, Dept. of Computer and Information Science, University of Pennsylvania, 1995.

[9] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, March 1995.

[10] Val Tannen. The monad of dons-trees, 1993. Personal Communication.

[11] J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.