A Query Language and Optimization Techniques for Unstructured Data*

Peter Buneman
University of Pennsylvania
peter@central.cis.upenn.edu

Susan Davidson†
University of Pennsylvania
susan@central.cis.upenn.edu

Gerd Hillebrand
University of Pennsylvania
gerdh@saul.cis.upenn.edu

Dan Suciu
AT&T Research
suciu@research.att.com

Abstract

A new kind of data model has recently emerged in which the database is not constrained by a conventional schema. Systems like ACeDB, which has become very popular with biologists, and the recent Tsimmis proposal for data integration organize data in tree-like structures whose components can be used equally well to represent sets and tuples. Such structures allow great flexibility in data representation

What query language is appropriate for such structures? Here we propose a simple language UnQL for querying data organized as a rooted, edge-labeled graph. In this model, relational data may be represented as fixed-depth trees, and on such trees UnQL is equivalent to the relational algebra. The novelty of UnQL consists in its programming constructs for arbitrarily deep data and for cyclic structures. While strictly more powerful than query languages with path expressions like XSQL, UnQL can still be efficiently evaluated. We describe new optimization techniques for the deep or "vertical" dimension of UnQL queries. Furthermore, we show that known optimization techniques for operators on flat relations apply to the "horizontal" dimension of UnQL.

1 Introduction

There are two good reasons to query and manipulate data whose structure is not constrained by a schema. First, some systems and proposals have recently emerged in which the schema is either absent or, when it exists, only loosley constrains the data; second, for the purposes of browsing it may be convenient to forget the schema, even though one exists. In this new "unstructured" approach to data representation each component, or object, is interpreted dynamically and may be linked to other components in an arbitrary fashion. We represent such data in a very general framework, which is roughly speaking an edge-labeled rooted directed graph. We call this model a "labeled tree" model because the query language we will develop is most easily thought of as a tree-traversing language, even though all expressible queries are well-defined on cyclic structures. To justify this choice of model let us briefly examine some current systems that can be conveniently represented in such a framework.

Unstructured data models. Biological data storage poses a problem for "fixed schema" systems, because the rapid evolution of experimental techniques requires constant adjustment of the schema [GRS93]. It is also convenient to have a model that accommodates missing data. One such database system that is extremely popular within the molecular biology community is ACeDB (A C. elegans Database) [TMD92]. Although ACeDB has a schema, it imposes only weak constraints on the database. A class in an ACeDB schema can

*University of Pennsylvania, Computer and Information Science Department, Technical Report MS-CIS 96-09. An extended abstract of this work appears in SIGMOD Proceedings, 1996.

†Corresponding author. Address: Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104-6389. Phone: (215) 898-3490. Fax: (215) 898-0587.

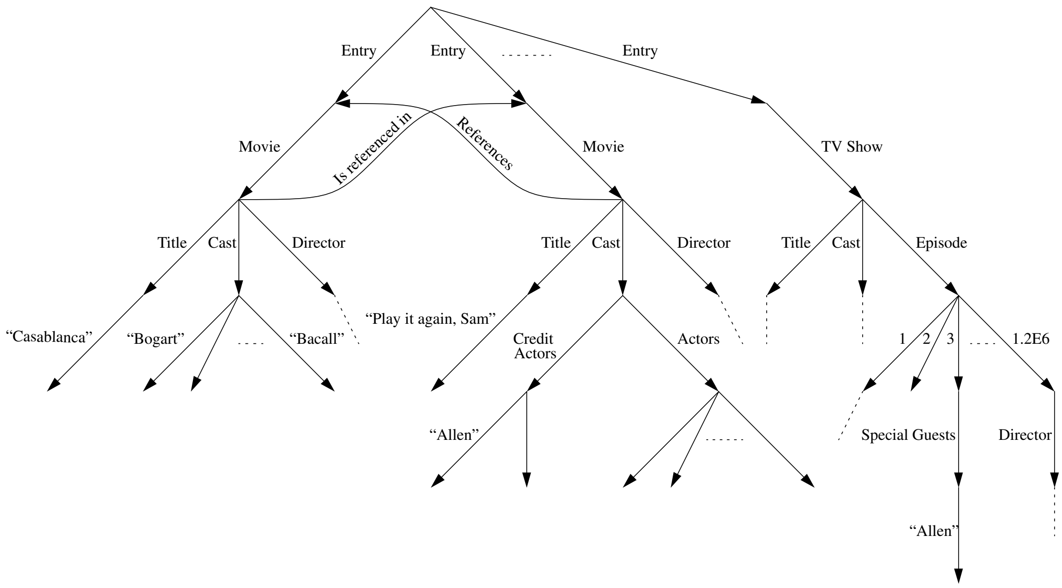


Figure 1: An example movie database.

be thought of as a labeled tree with non-terminal edges labeled by attribute names or base types and leaves labeled by base types or by other class names. Superficially, this is like an object-oriented model, and one could think of adapting an object-oriented query language. In fact, one of the languages we shall describe has some syntactic similarity to object-oriented query language such as OQL[?]; however the interpretation is very different. The query language for ACeDB allows selections of objects and pointer traversals, but it cannot perform general projections or joins.[1]

Another system that uses a tree-like model is Tsimmis [PGMW95], which has been proposed for heterogeneous data integration. In Tsimmis there is no schema, and the "type" of data is interpreted by the user from labels in the structure. In particular, the difference between a record and a set in a Tsimmis structure is that within a record node the edge labels are all distinct, whereas within a set node the edge labels are all the same. A query language has been proposed for Tsimmis, but it allows only limited forms of "deep" traversal of the data structures. Languages for restructuring Tsimmis would also appear to be important, for Tsimmis is proposed as a model for data exchange. Moreover optimization of such languages appears to be an open issue.

Browsing. Even if a schema exists for the data (e.g., data stored in a relational database management system), it may be convenient to ignore it for browsing purposes. For example, one might want to search all the character strings in the database to determine which tables contain a particular term. One cannot write a generic relational algebra expression to express such a query. Languages such as XSQL [KKS92] provide a solution to this, and these can be extended, with some limitations, to object-oriented databases. However the problem of "deep" queries and query optimization remains. An example of a "deep" query is known to most programmers: find a file which is somewhere in a tree-structured file system. Utilities to perform this particular query are usually available, but they do not generalize easily, and dealing with recursive structures (which can arise from the presence of symbolic links) is usually achieved through a hack.

Edge-labeled trees. As an example of the structure we want to deal with, consider the movie database shown in Figure 1.[2] The first two subtrees out of the root are both movies, but they differ in how they represent actors. The third subtree represents a TV series and has very little structure in common with

[1] The recent TableMaker extension to ACeDB provides projections and joins, but it is a hack and does not form a compositional query language.

[2] With thanks to *The Internet Movie Database* (to be found at `http://www.msstate.edu/Movies/`). The deliberately irregular example used here in no way reflects the details of that well-structured and very useful database.

movies. Furthermore, movies/TV series refer to each other, creating cycles in the structure. The situation is quite similar to that of a bibliographical database (cf. the examples given in [PGMW95]), where information about heterogeneous documents and their referencing relationship is represented in a graph-like structure.

Note that information only resides at labels. To express a structure in which information also resides at nodes (both ACeDB and Tsimmis allow this), we simply "migrate" that information down to a new edge attached to that node. In our examples we shall typically use various kinds of edge labels: character strings, integers, reals, booleans, and other base types together with symbols — the latter are also represented as strings but are used to represent what would be attribute names in a conventional database. Atomic data values such as strings and integers may occur anywhere in the tree, not just at terminal edges. This is similar to ACeDB, which allows integer labeled fields.

Outline of the paper. In the following sections we first develop the labeled tree data model and discuss correspondences with relational and nested relational databases. We then develop the query language UnQL (for Unstructured Query Language) through examples. On fixed-depth structures UnQL has the expressive power of the nested relational algebra. However, it can also express traversals of a tree of arbitrary depth, and these searches can be controlled by the use of regular expressions on paths. So far, this language is still limited in that it cannot *restructure* the database at arbitrary depths. We then give a more general—and more complex—construct that can restructure the database at arbitrary depths. After summarizing the UnQL constructs, we give a calculus into which our languages can be translated and provide optimization techniques in this calculus. In particular we show that "horizontal" optimizations for relational structures can be used and that these have "vertical" counterparts that apply to deep queries. We conclude with some questions about the connection between UnQL and other languages.

2 The Data Model

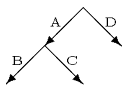
Labeled graphs Formally, our data model consists of *rooted, labeled graphs.* The labels carry all the basic information in this model, so we need to specify the types of data that can be used as labels. We assume that the usual base types *String*, *Integer*, *Real*, etc., are available. These are the types that would be used as values in a relation. In addition we shall use a new type *Symbol* for labels that would correspond to attribute names in a relation. We write numbers and symbols literally (the latter usually capitalized) and use quotation marks for strings, e.g., "*cat*". In what follows we make the simplifying assumption that labels can be symbols, strings, integers, etc; in fact, the type of labels is just the discriminated union of these base types.

Formally, a rooted, labeled graph is $G = (V, E, s, t, r, \lambda, \rho)$, where (V, E, s, t) is a multi-graph ($s : E \to V$ and $t : E \to V$ denote the source and the target of each edge), $r \in V$ is the root, and $\lambda : E \to \mathit{Label} \cup \{\varepsilon\}$ is the labeling function; ε and ρ are described below. We will denote with $x \xrightarrow{a} y$ an edge e going from x to y and labeled a (that is $s(e) = x, t(e) = y, \lambda(e) = a$). We will follow tree terminology: thus, the *children* of $v \in V$ are the nodes v' s.t. $\exists a.v \xrightarrow{a} v'$, and a *leaf* is a node v with no children.

Trees Alternatively, we provide a concise *notation* for rooted, labeled graphs. For the moment let us restrict ourselves to structures without cycles, and give the following explicit syntax for the construction of edge-labeled trees:

- $\{\}$ — the empty tree,
- $\{l \Rightarrow T\}$ — the tree whose root has an outgoing edge labeled l attached to the subtree t,
- $T_1 \cup T_2$ — the union of trees T_1 and T_2 formed by coalescing the roots of T_1 and T_2.

We shall also use the syntactic sugar $\{l_1 \Rightarrow T_1, l_2 \Rightarrow T_2, \ldots, l_n \Rightarrow T_n\}$ for $\{l_1 \Rightarrow T_1\} \cup \{l_2 \Rightarrow T_2\} \cup \cdots \cup \{l_n \Rightarrow T_n\}$. Each notation uniquely defines a rooted, labeled tree. E.g. $\{A \Rightarrow \{B \Rightarrow \{\}, C \Rightarrow \{\}\}, D \Rightarrow \{\}\}$ denotes the tree:



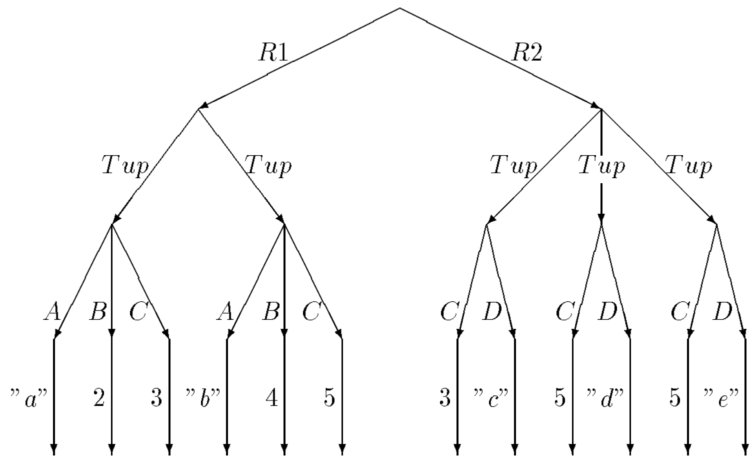
We can further simplify the syntax by shortening the notation for terminal edges, $l \Rightarrow \{\}$, to l, and by omitting the braces around singleton trees $\{l \Rightarrow T\}$. Our example then becomes $\{A \Rightarrow \{B, C\}, D\}$.

$R1$

A	B	C
"a"	2	3
"b"	4	5

$R2$

C	D
3	"c"
5	"d"
5	"e"



$$\{R1 \Rightarrow \{Tup \Rightarrow \{A \Rightarrow \text{"}a\text{"}, B \Rightarrow 2, C \Rightarrow 3\}, Tup \Rightarrow \{A \Rightarrow \text{"}b\text{"}, B \Rightarrow 4, C \Rightarrow 5\}\},$$
$$R2 \Rightarrow \{Tup \Rightarrow \{C \Rightarrow 3, D \Rightarrow \text{"}c\text{"}\}, Tup \Rightarrow \{C \Rightarrow 5, D \Rightarrow \text{"}d\text{"}\}, Tup \Rightarrow \{C \Rightarrow 5, D \Rightarrow \text{"}e\text{"}\}\}\}$$

Figure 2: Representations of a relational database.

Relational and nested relational databases. Relational databases are easily encoded as trees. Starting with tuples, if a relation has attributes $A_1, A_2, \ldots, A_n$, a tuple $\langle A_1 : v_1, A_2 : v_2, \ldots, A_n : v_n\rangle$ may be encoded as $\{A_1 \Rightarrow v_1, A_2 \Rightarrow v_2, \ldots, A_n \Rightarrow v_n\}$. Next, choosing a special label "Tup" to indicate the encoding of a tuple, we can encode a relation as $\{Tup \Rightarrow t_1, Tup \Rightarrow t_2, \ldots, Tup \Rightarrow t_n\}$, where $t_1, t_2, \ldots, t_n$ are the encodings of the tuples in the relation. Finally, the database itself is a set of relations named $R_1, R_2, \ldots, R_n$, and the encoding of the database is $\{R_1 \Rightarrow r_1, R_2 \Rightarrow r_2, \ldots, R_n \Rightarrow r_n\}$, where $r_1, r_2, \ldots, r_n$ are the encodings of the associated relations. For example, Figure 2 shows a simple relational database, first presented conventionally, then as a tree, and then in this encoding. Nested relational databases can be encoded similarly.

Syntax for cyclic structures. We extend now our syntax for trees so that we can tie the leaf nodes back into the tree. For this we allow ourselves to label some leaves with *tree markers*, $X_1, X_2, \ldots$. We denote with *Marker* the infinite set of all markers. In the graph model, the labeling of leaves with markers is captured by ρ, which is a partial function $\rho : V \to \mathit{Marker}$, mapping some of the leaves to markers. In the tree syntax, we allow a marker to stand for a tree. E.g. the trees $\{a \Rightarrow \{b, c \Rightarrow X\}, b \Rightarrow X, d\}$ and $\{a \Rightarrow X\} \cup Y$ "denote" the graphs in Figure 3 (a) and (b) respectively. Note that in order to represent the second tree we need the special symbol ε whose intuitive meaning is that of "no label"; more on it below. Finally, we extend our tree syntax with a construction for cyclic structures:

T where $X_1 = T_1, X_2 = T_2, \ldots, X_n = T_n$.

This expression denotes a (possibly cyclic) "tree". In it $T, T_1, T_2, \ldots, T_n$ are tree expressions built up from the previous tree constructors. The meaning is that of a graph obtained by "collapsing" all leaves labeled with some marker X_i with the root of e_i. For example,

X_1 where $X_1 = \{A \Rightarrow X_2, C \Rightarrow \{D, E \Rightarrow X_1\}\}$,
$X_2 = \{B \Rightarrow X_2\}$

denotes the cyclic structure depicted in Figure 4 (a). To simplify the formalism, instead of "collapsing" nodes we formally define the meaning of the where syntax in terms of ε-edges: namely we draw an ε-edge from every leaf labeled X_i to the root of T_i, for $i = 1, n$. We shall continue to use the term "tree" to describe such structures, understanding that they are really rooted directed connected graphs.

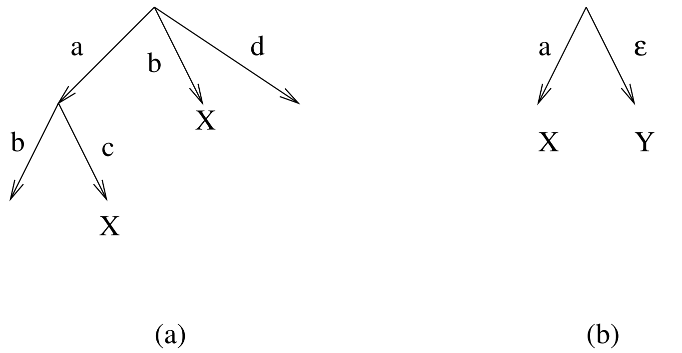


Figure 3: Graphs with markers.

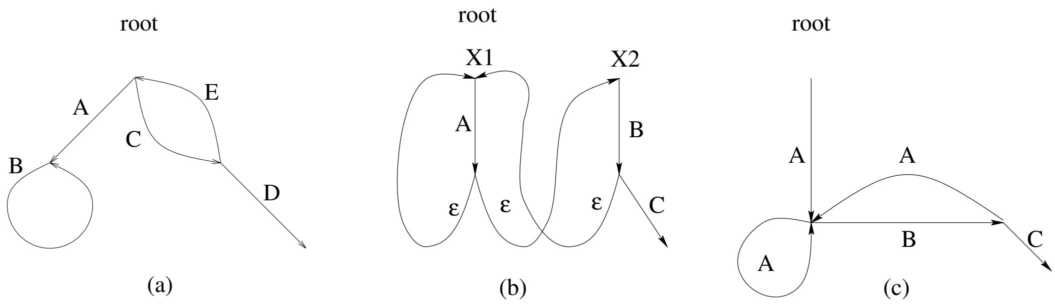


Figure 4: Cyclic structures.

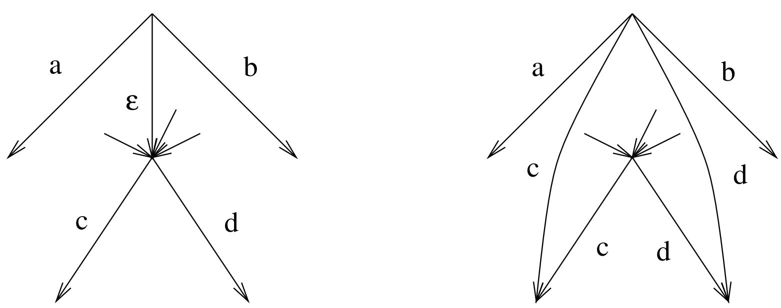


Figure 5: Meaning of ε-edges.

ε-edges and bisimulation ε-edges are a convenient way to draw some graphs, like those arising from cyclic definitions. Intuitively, their meaning is as folows: an ϵ-edge from node v to node w means that a copy of every edge leaving from w should leave from v too. E.g. the two graphs in Figure 5 should be considered "equal". Equality is captured precisely by the notion of *bisimulation*. Here $x \xrightarrow{\varepsilon^*} y$ denotes a path of the form $x \xrightarrow{\varepsilon} x_1 \xrightarrow{\varepsilon} x_2 \ldots \xrightarrow{\varepsilon} x_n \xrightarrow{a} y$.

Definition 2.1 *Two graphs $G_1 = (V_1, E_1, s_1, t_1, r_1, \lambda_1, \rho_1)$ and $G_2 = (V_2, E_2, s_2, t_2, r_2, \lambda_2, \rho_1)$ are bisimilar, iff there exists a relation $R \subseteq V_1 \times V_2$ such that:*

1. *$(r_1, r_2) \in R$.*
2. *Whenever $x_1, x_2 \in R$ and there exists a path $x_1 \xrightarrow{\varepsilon^*}\xrightarrow{a} y_1$ in G_1, with $a \neq \varepsilon$, then there exists a node y_2 and a path $x_2 \xrightarrow{\varepsilon^*}\xrightarrow{a} y_2$, in G_2 such that $(y_1, y_2) \in R$.*
3. *Similarly, with the roles of G_1 and G_2 reversed.*
4. *Whenever $x_1, x_2 \in R$ and there exists a path $x_1 \xrightarrow{\varepsilon^*} y_1$ in G_1 with $\rho_1(y_1) = X$, then there exists a node y_2 in G_2 and a path $x_2 \xrightarrow{\varepsilon^*} y_2$ with $\rho_2(y_2) = X$, and $(y_1, y_2) \in R$. Similarly, with the roles of G_1, G_2 reversed.*

For example, consider the following, rather tricky cyclic structure[3]: X_1 where $X_1 = \{A \Rightarrow (X_1 \cup X_2)\}, X_2 = \{B \Rightarrow (\{C\} \cup X_1)\}$. We draw it, rather naturally, as in Figure 4 (b): note that this is a graph without markers. Furthermore it is easy to see that this graph is bisimilar, hence "equal", to that in Figure 4 (c).

Ignoring markers for a moment, bismilarity is related to, but not identical, to the "weak bimsimilarity" used in process algebra [Mil89], where the "silent transitions" (τ) correspond to our ε edges. Namely two graphs G_1, G_2 are "weakly bisimilar" [Mil89] if there exists some relation R as in Definition 2.1, but in which item 2 is replaced with: whenever $x_1, x_2 \in R$ and there exists a path $x_1 \xrightarrow{\varepsilon^*}\xrightarrow{a}\xrightarrow{\varepsilon^*} y_1$ in G_1, with $a \neq \varepsilon$, then there exists a node y_2 and a path $x_2 \xrightarrow{\varepsilon^*}\xrightarrow{a}\xrightarrow{\varepsilon^*} y_2$, such that $(y_1, y_2) \in R$. Here we adopted our flavor of bisimulationa, because the weak bisimulation is not what we want. For example, the graphs in Figure 6 are bisimilar but not weakly bisimilar. Conversely however, one can prove that two weakly bisimilar graphs are bisimilar: this is not immediately obvious, but requires a small trick. Namely suppose that G_1, G_2 are weakly bisimilar, and let R be the largest weak bisimilarity relation [Mil89]. To show that R is also a bimisimilarity relation, let $(x_1, x_2) \in R$, and $x_1 \xrightarrow{\varepsilon^*}\xrightarrow{a} y_1$ in G_1. By weak bisimilarity we obtain nodes y_2, z_2 in G_2 such that $x_2 \xrightarrow{\varepsilon^*}\xrightarrow{a} y_2 \xrightarrow{\varepsilon^*} z_2$, and $(y_1, z_2) \in R$. However we need to show that $(y_1, y_2) \in R$. We show that $\{(y_1, y_2)\} \cup R$ is a weak bisimilarity relation too, hence $(y_1, y_2) \in R$. It is easy to show one of the two implications of weak bisimilarity, namely that for any $y_1 \xrightarrow{\varepsilon^*}\xrightarrow{b}\xrightarrow{\varepsilon^*} u_1$ there exists some $y_2 \xrightarrow{\varepsilon^*}\xrightarrow{b}\xrightarrow{\varepsilon^*} u_2$ with $(u_1, u_2) \in R$. For the converse, we reverse the roles of G_1, G_2, and show first that there exists some z_1 in G_1 s.t. $y_1 \xrightarrow{\varepsilon^*} z_1$ and $(z_1, y_2) \in R$.

Finally, we note that, in the absence of markes, ε edges can be completely eliminated.

Proposition 2.2 *Let G be a marker-free labeled, rooted graph. Then there exists a labeled, rooted graph G' bisimilar to G, and without ε edges.*

Proof: G' will have the same vertices as G, and will have edges $x \xrightarrow{a} y$ in G' for every path $x \xrightarrow{\varepsilon^*}\xrightarrow{a} y$ in G. Obviously G, G' are bisimilar. □

See [Kos95] for more comments of bisimilarity for data models.

Object Identities. Tree markers should not be confused with object identities, as different expressions may describe the same tree. For example all of the following three expressions denote the same tree:

X_1 where $X_1 = \{A \Rightarrow X_1\}$

X_1 where $X_1 = \{A \Rightarrow \{A \Rightarrow X_1\}\}$

X_1 where $X_1 = \{A \Rightarrow X_2\}, X_2 = \{A \Rightarrow X_1\}$

[3] Recall that markers may be used wherever trees may be used: hence, $X_1 \cup X_2$ is a valid tree expression.

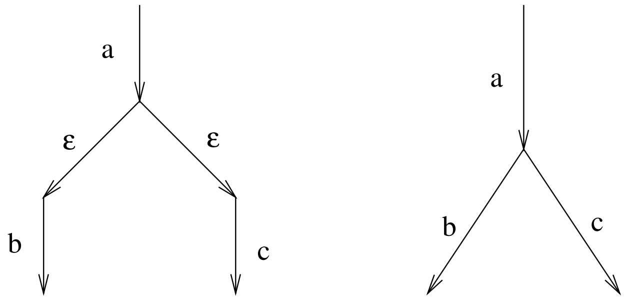


Figure 6: Two bisimilar graphs which are not weakly bisimilar.

Types. The type of cycle-free labeled trees has a simple description. Let *Label* be the type of edge labels and, for any type τ, let $\mathcal{P}_{\text{fin}}(\tau)$ describe the type of finite sets of τ. The type for labeled trees *Tree* satisfies the equation $\mathit{Tree} = \mathcal{P}_{\text{fin}}(\mathit{Label} \times \mathit{Tree})$.

Informally, this equation says that a tree is a set of pairs of labels and trees. Following [BTS91], we can obtain a natural form of computation for this type based on structural recursion. An initial version of this idea was pursued in [BDS94]. This paper describes new languages that are more general restrictions of structural recursion and are well-defined on cyclic structures; it also explores optimization techniques. In the development of these languages, it is important to remember that we are dealing with just two data types: *Label* and *Tree*.

3 UnQL: Selection Queries

We begin by presenting some examples of UnQL that query the tree down to some fixed depth. On data structures that represent relational databases, such queries are equivalent to relational algebra queries. We then give examples that query to an arbitrary depth, and retrieve information from anywhere in the tree. Even though such queries contain complex "path expressions," they are nevertheless meaningful for cyclic structures.

3.1 Scratching the Surface—the Top-Level Fragment of UnQL

Consider some simple examples on the relational database of Figure 2, which we assume is named DB.

Example 3.1 The expression

$\mathsf{select}\ t$

$\mathsf{where}\ R1 \Rightarrow \backslash t \leftarrow DB$

says "compute the union of all trees t such that DB contains an edge $R1 \Rightarrow t$ emanating from the root." There is only one such edge in Figure 2; this query therefore simply returns the set of tuples in $R1$. The returned expression is:

$\{\ Tup \Rightarrow \{A \Rightarrow "a", B \Rightarrow 2, C \Rightarrow 3\},$

$\quad Tup \Rightarrow \{A \Rightarrow "b", B \Rightarrow 4, C \Rightarrow 5\}\}$

There are some important differences between our *comprehension syntax* [BLS+94] and that of relational calculus or languages derived from SQL. In the where part of the expression (there is no from component), the form $R1 \Rightarrow \backslash t \leftarrow DB$ is a *generator*, which has two components, a *pattern* $R1 \Rightarrow \backslash t$ (everything to the left of the "generator" ($\leftarrow$) arrow) and a tree DB (the right-hand side of the $\leftarrow$ arrow). A tree is simply a set of edge/subtree pairs, so the generator matches the pattern to each of these pairs. Only those edges labeled $R1$ will match. When such a match occurs, the variable $\backslash t$ is bound to the associated subtree, and the expression in the select clause is evaluated and added to the result.

When variables are introduced in UnQL, they are flagged with a backslash. They can then be used later (without the backslash) in the where part of the comprehension as well as in the select part. This explicit binding of variables is needed to avoid ambiguities that would arise in nested queries.

Example 3.2 The following expression uses a label variable $\backslash l$ to match any edge emanating from the root.

$\textsf{select}\ t$

$\textsf{where}\ \backslash l \Rightarrow \backslash t \leftarrow DB$

The result is the union of all tuples in both relations—a heterogeneous set that cannot be described by a single relation.

Example 3.3 Here we join $R1$ and $R2$ on their common attribute C and then project onto A and D:

$\textsf{select}\ \{Tup \Rightarrow \{A \Rightarrow x, D \Rightarrow z\}\}$

$\textsf{where}\ R1 \Rightarrow Tup \Rightarrow \{A \Rightarrow \backslash x, C \Rightarrow \backslash y\} \leftarrow DB,$

$\qquad R2 \Rightarrow Tup \Rightarrow \{C \Rightarrow y, D \Rightarrow \backslash z\} \leftarrow DB$

$R1 \Rightarrow Tup \Rightarrow \{A \Rightarrow \backslash x, C \Rightarrow \backslash y\}$ is a *tree pattern*. It matches any subtree of DB starting at the root, continuing with an $R1$ edge, then with a Tup edge which has two edges labeled A and C respectively. Moreover it binds x and y to the subtrees after A and C. Note that the variable y is bound in the pattern of one generator and then used as a constant in the pattern of the second.

Example 3.4 This query performs a group-by operation on $R2$ along the C column:

$\textsf{select}\ \{x \Rightarrow (\textsf{select}\ y\ \textsf{where}$

$\qquad\qquad R2 \Rightarrow Tup \Rightarrow \{C \Rightarrow x, D \Rightarrow \backslash y\} \leftarrow DB)\}$

$\textsf{where}\ R2 \Rightarrow Tup \Rightarrow C \Rightarrow \backslash x \Rightarrow \{\} \leftarrow DB$

The use of $\backslash x \Rightarrow \{\}$ is needed to bind x to an edge label rather than a tree, so that the expression $x \Rightarrow \ldots$ in the select clause makes sense. In contrast, $\backslash y$ ranges over trees. The output from this last query is the tree $\{3 \Rightarrow \{"c"\}, 5 \Rightarrow \{"d", "e"\}\}$.

Example 3.5 Turning to the movie database in Figure 1, we see that there are some queries that can be answered by the techniques we have already developed. For example, give the titles and casts of all entries—movies, TV shows, etc.

$\textsf{select}\ \{Tup \Rightarrow \{Title \Rightarrow x, Cast \Rightarrow y\}\}$

$\textsf{where}\ Entry \Rightarrow \_ \Rightarrow \{Title \Rightarrow \backslash x, Cast \Rightarrow \backslash y\} \leftarrow DB$

Here the "wildcard" symbol _ matches any edge label; its use is equivalent to binding a fresh anonymous variable at that point. Also note that the query has returned a tree that is a set of pairs, but the components of these pairs (x and y) are trees.

Example 3.6 The next example is more problematic. We want a binary relation consisting of actress/actor and title tuples for movies. The problem is that the information is not uniformly located within the tree.

$\textsf{select}\ \{Tup \Rightarrow \{Actor \Rightarrow x, Title \Rightarrow y\}\}$

where

$\quad Entry \Rightarrow Movie \Rightarrow \{Title \Rightarrow \backslash y, Cast \Rightarrow \backslash z\} \leftarrow DB,$

$\quad \backslash x \Rightarrow \{\} \leftarrow z \cup (\textsf{select}\ u\ \textsf{where}\ \_ \Rightarrow \backslash u \leftarrow z),$

$\quad isstring(x)$

The assumption we have made is that the names we want will be found immediately below the $Cast$ edge or one step further down. Moreover we only want those edges that are strings. This query illustrates the use of a *condition*, which is the second construct that can occur in the where part of a comprehension. We also assume that the type of labels is a discriminated union, so properties such as *isstring* should be available.

3.2 Taking the Plunge: Deep Queries.

It is apparent from the previous examples that we need more expressive power if we are to look for data whose depth in the tree is not predetermined by any schema.

Example 3.7 Perhaps the simplest of all queries that look arbitrarily deep into the database is to find all edges with a certain property. For example, to find the set of all strings in the database:

select $\{l\}$

where $\_* \Rightarrow \backslash l \Rightarrow \_ \leftarrow DB, isstring(l)$

Here the $\_*$ is a "repeated wildcard" that matches any path, i.e., sequence of edges, in the tree. Such a construct is proposed in [PGMW95]. However, we shall find some useful queries where we need more than this, i.e. we need to specify regular expressions on paths. For this, we adopt a `grep`-like syntax.

One may wonder whether queries containing a $\_*$ construct are well defined on cyclic structures, given that the number of paths in such structures is infinite. However, since cyclic structures have only a finite number of distinct subtrees, there is only a finite number of distinct assignments of labels and trees to the variables in the where clause, so the output of the query is still finite.

The use of a leading $\_*$ is so common that we shall use a special abbreviation $p \twoheadleftarrow t$ for $\_* \Rightarrow p \leftarrow t$ for any pattern p and any tree expression t. The query above then becomes:

select $\{l\}$

where $\backslash l \Rightarrow \_ \twoheadleftarrow DB, isstring(l)$

Example 3.8 Here we use consecutive "deep" generators to find all the movies involving "Bogart" and "Bacall":

select $\{Movie \Rightarrow x\}$

where $Movie \Rightarrow \backslash x \twoheadleftarrow DB,$

$\quad "Bogart" \Rightarrow \_ \twoheadleftarrow x,$

$\quad "Bacall" \Rightarrow \_ \twoheadleftarrow x$

This will find all movie edges in the database (no matter where they are) and return those edges, with their subtrees, that contain both "Bogart" and "Bacall" somewhere beneath them.

Example 3.9 A problem with the example above is that it may return more than is required. For example, a movie will be returned if it *refers* to a movie in which Bogart and Bacall are involved. In order to avoid such paths we use a more complicated pattern in a path:

select $\{Movie \Rightarrow x\}$

where $Movie \Rightarrow \backslash x \twoheadleftarrow DB,$

$\quad [\hat{}Movie]* \Rightarrow "Bogart" \Rightarrow \_ \leftarrow x,$

$\quad [\hat{}Movie]* \Rightarrow "Bacall" \Rightarrow \_ \leftarrow x$

Following `grep`, the pattern $[\hat{}Movie]*$ matches any path that does not contain the label $Movie$. Arbitrary regular expressions may be used on labels.

4 UnQL: Restructuring Queries

All the queries discussed so far have only exploited depth by pulling out subtrees. In this section we describe the traverse query construct, whose output tree can represent arbitrarily deep changes in the input tree. Consider first a simple example:

Example 4.1 Replace all $SpecialGuest$ labels with a $Featuring$ label:

traverse DB giving X

case $SpecialGuest \Rightarrow \_$ then $X := \{Featuring \Rightarrow X\}$

case $\backslash l \Rightarrow \_$ then $X := \{l \Rightarrow X\}$

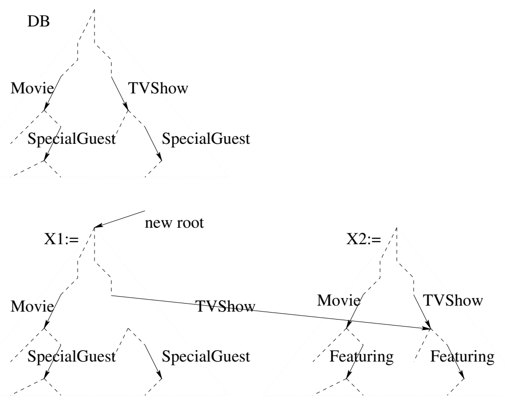


Figure 7: Replacing all *SpecialGuest* edges underneath *TVShow*s. The new root is the root of the first tree. Note that part of both trees are inaccessible from this root: they need not be constructed during query evaluation.

The construction says that we have to replace every edge of the tree DB with some tree as follows. Whenever we see an edge labeled *SpecialGuest* in DB, we replace it with the tree $\{Featuring \Rightarrow X\}$: in essence this is just an edge, leading to a marker X. Whenever we see an edge labeled something else, like l, we replace it with the tree $\{l \Rightarrow X\}$. The X's of the different trees are now joined as follows: each X-leaf of some tree replacing an edge e is joined with the roots of the trees replacing the successors of e. The order in which we write the two case statements matters: had we reversed them, we would have produced a tree identical to the original one.

The restructuring performed by the traverse construct is relatively simple, in that it acts only locally—on edges. This simplicity allows it to have meaning on cyclic structures (cycles will simply be transformed into other cycles), but limits its expressive power. We can increase its expressiveness significantly by allowing several markers instead of one, while still confining ourselves to local restructurings, as illustrated in the following example.

Example 4.2 Suppose we want to replace every edge *SpecialGuest* with a *Featuring* edge, but only in *TVShow*s, and leave the *SpecialGuest* edge for other kinds of movies untouched. Being part of a *TVShow* is not local information available at that edge, and in general we don't know how far apart the *TVShow* edge is from its underlying *SpecialGuest*. The trick here is to construct two trees in parallel, instead of one, having edges going from the first to the second one, as illustrated in Figure 7. The first tree, t_1, will be an identical copy of the input database DB. The second tree, t_2, will be a copy of DB with all *SpecialGuest* edges replaced by *Featuring*. In addition, every edge labeled *TVShow* in t_1 will be redirected to the corresponding node in t_2. As a consequence, every edge in DB will be replaced by two "small" trees, and we will use two markers X_1 and X_2 to keep track of them:

traverse DB giving X_1, X_2
  case $TVShow \Rightarrow \_$ then $X_1 := \{TVShow \Rightarrow X_2\}$
  case $SpecialGuest \Rightarrow \_$ then $X_2 := \{Featuring \Rightarrow X_2\}$
  case $\backslash l \Rightarrow \_$ then $X_1 := \{l \Rightarrow X_1\}$,
    $X_2 := \{l \Rightarrow X_2\}$

In this construction, pattern matching is done twice for each edge, once for X_1 and once for X_2. Clauses that do not provide an assignment to the marker under consideration are disregarded. Thus, on a *TVShow* edge, the first clause would succeed for X_1 and the last clause for X_2.

By convention, the final result is the tree rooted at the first marker X_1, that is, t_1. Parts of t_1 and t_2 will be superfluous, like the edges in t_1 beneath a *TVShow* edge, and those in t_2 above a *TVShow* edge. Since they are inaccessible from the root they can be discarded; in fact, during the execution of this query those parts of t_1 and t_2 need not even be constructed.

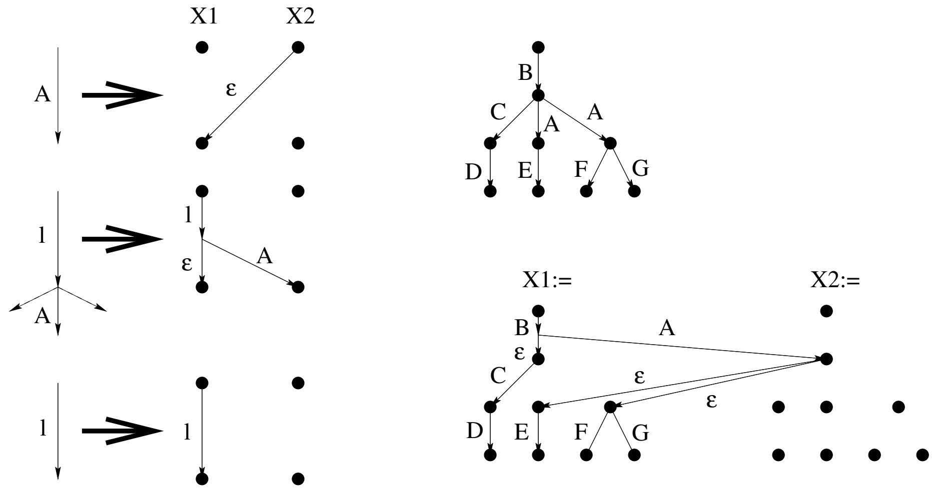


Figure 8: The "deep nest" query of Example 4.3

Example 4.3 Consider the following deep-nest example: in a given tree we want to nest all edges labeled A. That is, every subtree of the form $\{A \Rightarrow t_1, A \Rightarrow t_2, \ldots, A \Rightarrow t_n\} \cup t$, where t has no top-most A edges, has to be replaced with $\{A \Rightarrow t_1 \cup \ldots \cup t_n\} \cup t$. The query is given below: for simplicity we assume that no A-edges occur at the topmost level and that there are no two consecutive A-edges in the tree. See Figure 8 for an illustration.

traverse DB giving X_1, X_2

case $A \Rightarrow \_$ then $X_1 := \{\}, X_2 := X_1$

case $\backslash l \Rightarrow A \Rightarrow \_$ then $X_1 := \{l \Rightarrow (X_1 \cup \{A \Rightarrow X_2\})\}$,

$X_2 := \{\}$

case $\backslash l \Rightarrow \_$ then $X_1 := \{l \Rightarrow X_1\}, X_2 := \{\}$

traverse and select are typically executed by traversing the tree and, in the process, constructing the resulting tree. When two such queries are composed, it might appear that one has to traverse the whole tree, build an intermediate result, and then traverse the result. However, we show in Section 6 that the composed query can be optimized: we "fuse" together the two traversals into a single one and eliminate the intermediate result. For example, recall the query in Example 4.2 returning a tree T with some of its *SpecialGuest* edges replaced by *Featuring* edges. Now assume that we want to compose this query with: select $\{SpecialGuest \Rightarrow t\}$ where $SpecialGuest \Rightarrow \backslash t \leftarrow T$. By composing the two queries we obtain the set of all *SpecialGuest*s, but excluding those from *TVShows*. Of course this can be easily expressed as a single select query. We will show in Section 6 how this single query can be derived from simple algebraic equations.

5 Summary of UnQL

For completeness, we briefly enumerate UnQL's operators: $\{\}$, $\{\_ \Rightarrow \_\}$, $\cup$, @, if _ then _ else, select, traverse. We have already seen examples of all these with the exception of @, which is described in Section 6. Here we summarize the syntax for select and traverse. In general, a select expression has the form:

select E

where $C_1, C_2, \ldots, C_m$

E is another UnQL expression and each C_i is either a *generator* or a *condition*. Conditions are predicates on labels such as $l_1 = l_2$ or $isstring(l)$, and tree emptiness tests like $isempty(t)$. Generators are of the form $P \leftarrow T$, where P is a pattern and T is a UnQL expression. A pattern P is either (1) a tree constant or a tree variable, or (2) $\{R_1 \Rightarrow P_1, \ldots, R_m \Rightarrow P_m\}$ with each P_i a pattern, and each R_i either a variable $\backslash l$, or a regular expression. We abbreviate $\{R_1 \Rightarrow P_1\}$ with $R_1 \Rightarrow P_1$. Regular expressions may use, but not bind variables, hence we can have $\backslash l \Rightarrow \backslash t \leftarrow DB$ and $l* \Rightarrow \backslash t \leftarrow DB$, but not $(\backslash l)* \Rightarrow \backslash t \leftarrow DB$. Some examples of more complex patterns are: $(\_ \Rightarrow \_)* \Rightarrow \backslash l \Rightarrow \backslash t \leftarrow DB$ binds l to an edge label which is at an even depth in DB. $[\hat{}TVShow]* \Rightarrow \backslash l \Rightarrow [\hat{}SpecialGuest]* \Rightarrow l \Rightarrow \backslash t \leftarrow DB$ binds l to an edge label which occurs at least twice in DB, first before any $TVShow$ edge and second before any $SpecialGuest$ edge.

The general form of a traverse expression is:

traverse T giving $X_1, \ldots, X_k$
  case P_1 [where C_1] then $X_{i_1} := e_1$
  $\vdots$ $\vdots$
  case P_m [where C_m] then $X_{i_m} := e_m$

Here $X_1, \ldots, X_k$ are tree markers, $P_1, \ldots, P_m$ are patterns starting with an edge (i.e., of the form $l \Rightarrow P$ or $\backslash x \Rightarrow P$), $C_1, \ldots, C_m$ are conditions (they are optional), and the e_i are tree expressions which may involve the markers $X_1, \ldots, X_k$. As illustrated in the previous example, we may group several then clauses into one, when they share the same pattern and the same condition.

Briefly, the semantics of traverse is as follows. Each edge of the input tree T will replaced with k trees $t_1, \ldots, t_k$, corresponding to the markers $X_1, \ldots, X_k$, with lots of ε edges pointing between them. We explain how to construct t_i, for $i = 1, k$. We start by making fresh copies, call them v_i, of each vertice v of T. Let $x \xrightarrow{a} y$ be some edge in T, and let t be the subtree of T rooted at y. We match this edge successively against all patterns P_j (and corresponding conditions C_j) that have X_i on their right, for $j = 1, \ldots, m$. We stop at the first match we find, say it is j (of course, the order in which we try to match matters). Then we evaluate e_j, with the bindings obtained during pattern matching, hence e_j may use the label a and the tree t; by convention, $e_j = \{\}$ if no match occured. Finally we "include" the resulting tree in t_i, add an ε edges from x_i to the root of e_j, and add ε edges from the markers $X_1, X_2, \ldots, X_k$ in e_j to the nodes $y_1, y_2, \ldots, y_k$ in the trees $t_1, t_2, \ldots, t_k$ respectively. Note that the latter action adds ε edges from t_i to all other trees. Finally, the result of the traverse expression is the tree "visible" from the root of t_1.

We end this section by proving that on relational databases UnQL collapses to relational algebra. Recall that the tree model naturally encodes relational databases as trees of depth 4 (Figure 2). Similarly, it encodes nested relational databases as trees of a fixed depth. On these inputs, UnQL has the same expressive power as relational algebra, or nested relational algebra.

Theorem 5.1 *On trees encoding relational databases, or nested relational databases, UnQL has exactly the same expressive power as relational algebra, or nested relational algebra respectively.*

We will prove this theorem in Subsection 6.5.

However, note that UnQL can be more succinct than relational algebra: the UnQL query

select $\{Tup \Rightarrow t\}$
where $\_ \Rightarrow Tup \Rightarrow t \leftarrow DB$,
  $\_ \Rightarrow "Smith" \leftarrow t$

selects all tuples containing the string "Smith" from the database. Ignoring for the moment the fact that the result is a heterogeneous set, one can translate this query into relational algebra, but there will be a *different* translation for each database schema: as the schema becomes more complex, so does the translated query.

6 A Calculus for UnQL and its Optimizations

We now show that UnQL is equivalent to a simple calculus, called UnCAL for *unstructured calculus*. Following [BBW92], we use the term *calculus* in the sense of *lambda calculus* (a formalism with variables and functions), rather than in the sense of *relational calculus* (a logic, with variables and quantifiers). We sketch a simple, effective procedure for translating UnQL queries into UnCAL expressions. By simplifying UnQL into UnCAL, we achieve two purposes. First, we can prove properties about UnQL, for example that all queries are defined even on cyclic structures and are computable in PTIME. Second, and more important, we derive optimization techniques for UnQL based on the algebraic identities that hold for UnCAL. In this section we show some of these equations and illustrate the corresponding optimizations.

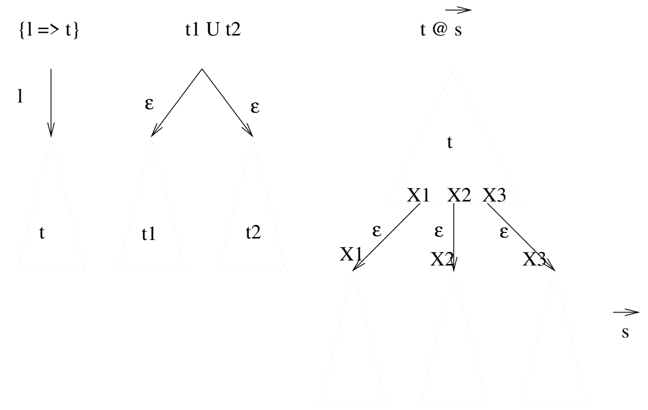


Figure 9: The semantics of $\{l \Rightarrow t\}, t_1 \cup t_2, t \ @_{\{X_1,X_2,X_3\}} \ \vec{s}$.

We will give two different semantics for the UnCAL language: one for labeled, rooted graphs, and the other for tree notations. For graphs, the main construct in UnCAL, *gext*, is described by a *parallel process*. We prove that the UnCAL queries do not break bisimilarity equivalence classes: any UnCAL query applied to two bisimilar graphs returns two bisimilar graphs (Proposition 6.1). For tree notations, the semantics of *gext* is described by a *recursive process* on tree notations. We show that the two semantics are equivalent.

6.1 UnCAL Syntax

The salient construct in UnCAL is its recursion operator *gext*, which is the mechanism underlying the traverse construct. Like traverse, this is a rather unorthodox operator, in that it uses tree markers in an essential way. While real-life examples probably will not put too much burden on these markers, the interactions between different sets of markers that result during the optimization steps are, as we shall see shortly, somewhat delicate. Hence UnCAL is quite picky about keeping track of the markers. For a finite set $\mathcal{X}$ of markers, we will denote by $Tree_{\mathcal{X}}$ the set of trees having only markers in $\mathcal{X}$. Then, $\mathcal{X} \subseteq \mathcal{X}'$ implies $Tree_{\mathcal{X}} \subseteq Tree_{\mathcal{X}'}$. Furthermore, for a reason that will become apparent shortly, we will assume that we can put together two markers to build a new one; that is, $X \cdot Y$ will be a new marker built from X and Y, and it will be distinct from all other markers built in this way. For two sets of markers $\mathcal{X}$ and $\mathcal{Y}$, $\mathcal{X} \cdot \mathcal{Y}$ denotes the set of markers $X \cdot Y$, with $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$. UnCAL's operators are listed in Figure 10. We will describe the semantics of UnCAL's operators on two models: first on the rooted, labeled graphs, and second on their syntactic notation given in Section 2.

6.2 UnCAL Semantics on Rooted, Labeled Graphs

The semantics of most of the UnCAL is obvious. Refering to Figure 10, $\{\}$ returns a graph with a single node, and no edges (representing the empty set). A marker X returns also a graph with one node, no edges, but in which the node is labeled with the marker X. $\{l \Rightarrow t\}$, $t_1 \cup t_2$, and $t \ @_{\mathcal{X}} \ \vec{s}$ are depicted in Figure 9. In an append expression, $t \ @_{\mathcal{X}} \ \vec{s}$, t is a tree of type $Tree_{\mathcal{X}}$ and $\vec{s}$ is an $\mathcal{X}$-indexed family of trees of type $Tree_{\mathcal{Y}}$. Such families occur often in UnCAL, and we write them as $\vec{s} = \{X_1 := s_1, \ldots, X_n := s_n\}$, when $\mathcal{X} = \{X_1, \ldots, X_n\}$; we denote the set of $\mathcal{X}$-indexed families of trees with $\mathcal{Y}$ markers with $Tree^{\mathcal{X}}_{\mathcal{Y}}$. Intuitively, the result of append is obtained by replacing each marker X_i in t with the tree s_i: formally this is done by drawing ε-edges from every leaf labeled X_i to the root of s_i, see Figure 9. The $\mathcal{Y}$ markers in the s_i are preserved in the output, which is therefore of type $Tree_{\mathcal{Y}}$.

As a simple example with only one marker, consider $t = \{A \Rightarrow B, C \Rightarrow X\} \cup X$ and[4] $s = \{B, D \Rightarrow E\}$. Then $t \ @_{\{X\}} \ s = \{A \Rightarrow B, C \Rightarrow \{B, D \Rightarrow E\}, B, D \Rightarrow E\}$. For an example with two markers, let $t = \{A \Rightarrow (X_1 \cup X_2), B \Rightarrow (\{C\} \cup X_1)\}$, and let $\vec{s}$ be the $\{X_1, X_2\}$-indexed family $\{X_1 := \{B, D\}, X_2 := \{B, D \Rightarrow E\}\}$. Then $t \ @_{\{X_1,X_2\}} \ \vec{s} = \{A \Rightarrow \{B, D, D \Rightarrow E\}, B \Rightarrow \{C, B, D\}\}$. See Figure 11 (a) and (b).

[4] Strictly speaking s is the $\{X\}$-indexed family $\{X := \{B, D \Rightarrow E\}\}$: we omit mentioning X when it is clear from the context.

$$\frac{}{\{\} : Tree_{\mathcal{X}}} \qquad \frac{X \in \mathcal{X}}{X : Tree_{\mathcal{X}}} \qquad \frac{l : Label \quad t : Tree_{\mathcal{X}}}{\{l \Rightarrow t\} : Tree_{\mathcal{X}}}$$

$$\frac{l_1 : Label \quad l_2 : Label}{l_1 = l_2 : Bool} \qquad \frac{t : Tree_{\mathcal{X}}}{isempty(t) : Bool}$$

$$\frac{b : Bool \quad t_1 : Tree_{\mathcal{X}} \quad t_2 : Tree_{\mathcal{X}}}{\textsf{if } b \textsf{ then } t_1 \textsf{ else } t_2 : Tree_{\mathcal{X}}}$$

$$\frac{t_1 : Tree_{\mathcal{X}} \quad t_2 : Tree_{\mathcal{X}}}{t_1 \cup t_2 : Tree_{\mathcal{X}}} \qquad \frac{t : Tree_{\mathcal{X}} \quad \vec{s} : Tree_{\mathcal{Y}}^{\mathcal{X}}}{t \mathbin{@_{\mathcal{X}}} \vec{s} : Tree_{\mathcal{Y}}}$$

$$\frac{f : Label \times Tree_{\mathcal{Y}} \to Tree_{\mathcal{X}}^{\mathcal{X}} \quad t : Tree_{\mathcal{Y}}}{gext_{\mathcal{X}}(f)(t) : Tree_{\mathcal{X}\cdot\mathcal{Y}}^{\mathcal{X}}}$$

Figure 10: The rules for UnCAL.

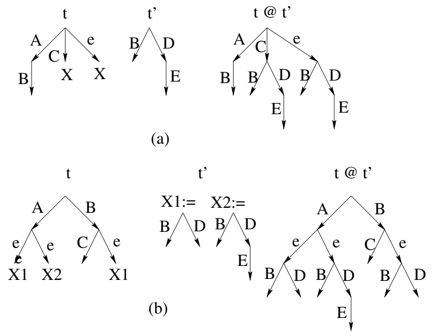


Figure 11: Two examples of append, $t \mathbin{@} t'$.

The $isempty(t)$ construct has a more complex semantics than it looks at a first glance: namely it is true iff there exists a path from the root of t to an edge labeled a, for some $a \neq \varepsilon$. This requires the computation of transitive closure on the subgraph of ε edges. As a consequence of this definition, a tree consisting of a single marker is empty, i.e. $isempty(X) = true$.

UnCAL's most complex operator is $gext_{\mathcal{X}}(f)(T)$. Here f is a function taking a label l and a tree t as inputs and returning an $\mathcal{X}$-indexed family of trees with markers $\mathcal{X}$, while the result is an $\mathcal{X}$-indexed family of trees. $gext_{\mathcal{X}}(f)(T)$ has the same meaning as a certain restricted form of traverse:

traverse T giving $\mathcal{X}$

case $\backslash l \Rightarrow \backslash t$ then $f(l, t)$

We will describe the semantics of $gext$ in detail next. Assume $\mathcal{X} = \{X_1, \ldots, X_k\}$. Then $f(l, t)$ is a family of k functions, $X_1 := f_1(l, t), \ldots, X_k := f_k(l, t)$. On rooted, labeled graphs, we define $gext$ as a *parallel process*. Namely to compute $T' = gext_{\{X\}}(f)(T)$, we proceed in three steps. (1) make k fresh copies $v_1, \ldots, v_k$ for every node v in the tree T. (2) For every non-ε edge $u \xrightarrow{a} v$ in T, construct k fresh trees, call them $T^i_{u \xrightarrow{a} u}$, $i = 1, k$ obtained by computing $f_i(a, t)$, where t is the subtree of T rooted at v; for every ε edge $u \xrightarrow{\varepsilon} v$ in T, let $T^i_{u \xrightarrow{\varepsilon} u}$ be a fresh tree consisting of a single ε edge with the leaf labled X_i. Here "fresh" means that trees constructed for different edges are disjoint, and they are disjoint from the nodes v_i constructed above. (3) For each edge $u \xrightarrow{a} v$, with $a \in Label$ or $a = \varepsilon$, and any $i = 1, k$, draw an ε edge from u_i to the root of $T^i_{u \xrightarrow{a} v}$; draw ε edges from every leaf X_j in $T^i_{u \xrightarrow{a} v}$ to v_j, $j = 1, k$; finally, whenever some leaf u was labeled with some marker Y in T, label the leaves $u_1, \ldots, u_k$ in T' with the markers $X_1 \cdot Y, \ldots, X_k \cdot Y$ respectively. By convention, the resulting rooted, labeled graph T' has root r_1, where r was the root of T.

Figure 12 offers an intuitive illustration for the case when $\mathcal{X}$ has a single marker. Obviously, $gext$ has meaning on cyclic structure: a cycle in T simply converts into a cycle in the result.

At this point we have given a complete definition of UnCAL in terms of rooted, labeled graphs with ε

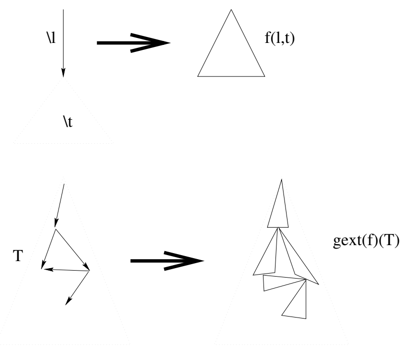


Figure 12: Visualization of $gext(f)(T)$: f is applied independently to each edge, and the resulting "small" trees are joined.

edges. But recall that we may have different graphs representing the same data, when they are bisimilar. Will we get different results if we apply some UnCAL query to two graphs representing the same data ? The following proposition shows that the answer is no:

Proposition 6.1 *Let G, G' be two bisimilar rooted, labeled graphs and e some UnCAL expression. Then the graphs $e(G)$ and $e(G')$ are bisimilar.*

Proof: We prove this by straightforward induction on the structure of e. We illustrate with 2 cases:

Union $e = e_1 \cup e_2$. By induction hypothesis $e_1(G), e_1(G')$ are bisimilar, and so are $e_2(G), e_2(G')$. By definition $e(G)$ has a root node with two ε edges leading to the roots of $e_1(G)$ and $e_2(G)$ respectively. It is easy to prove that this is bisimilar to $e(G')$.

Gext For illustration, assume $e(G) = gext_{\{X\}}(f)(G)$, i.e. we take *gext* on a single marker, and apply it directly to the input graph G. Recall the definition of the graph $e(G)$: it has a copy x_1 of every node x in G, and a fresh copy of the tree $f(a,t)$ for every edge $x \xrightarrow{a} y$ in G; similarly for $e(G')$. By hypothesis G, G' are bisimilar, say R is a bisimilarity relation between G, G': let us denote with R_0 its "copy" over nodes x_1, i.e. $R_0 = \{(x_1, x'_1) \mid (x, x') \in R\}$. Moreover, for every two edges $x \xrightarrow{a} y$ in G and $x' \xrightarrow{a} y'$ in G for which $(y, y') \in R$, by induction hypothesis the tree $f(a,t) = T_{x \xrightarrow{a} y}$ is bisimilar to $f(a,t') = T_{x' \xrightarrow{a} y'}$; let $R_{x \xrightarrow{a} y, x' \xrightarrow{a} y'}$ be a bisimilarity relation between the two trees (when $a = \varepsilon$, both trees have a single ε edge, and we take $R_{x \xrightarrow{a} y, x' \xrightarrow{a} y'}$ to be the isomorphism between them). Now we prove that the relation $\bar{R} \stackrel{\text{def}}{=} R_0 \cup R_1$, where

$$R_1 = \bigcup_{x \xrightarrow{a} y \in G, x' \xrightarrow{a} y' \in G', (y,y') \in R} R_{x \xrightarrow{a} y, x' \xrightarrow{a} y'}$$

is a bisimilarity relation between $e(G)$ and $e(G')$.

We only consider the case when we start with $(x_1, x'_1) \in R_0$ (the case when we start in R_1 is handled with minor changes), and say we have a path $x_1 \xrightarrow{\varepsilon^*} \xrightarrow{b} y$ in $e(G)$. Certainly the last edge, $z \xrightarrow{b} y$, is in one of the "small" trees, say $T_{u \xrightarrow{a} v}$, i.e. y is not of the form y_1. That is, the path has two parts: (1) it starts at x_1 and has an ε^* portion going through $x_1 = x_1^1, x_1^2, \ldots, x_1^k = u_1$, and (2) then has an ε edge from u_1 to the root of $T_{u \xrightarrow{a} v}$, from here follows an ε^* path to some z, and finally $z \xrightarrow{b} y$; the second part is entirely within $T_{u \xrightarrow{a} v}$ (exept for the first ε edge). We have to show that we can "replicate" the entire path in $e'(G)$.

Extend $x_1^1, x_1^2, \ldots, x_1^k = u_1$ by defining $x_1^{k+1} = v_1$. Between any two such nodes x_1^{i-1}, x_1^i there is a "little" tree which the path traverses (the last tree is traversed only partially), say $T_{x^{i-1} \xrightarrow{l^i} x^i}$. The roots of these trees give us a path $x^0 \xrightarrow{l^1} x^1 \xrightarrow{l^2} x^2 \ldots x^k \xrightarrow{l^{k+1}} x^{k+1}$ in G, where $l^{k+1} = a$ from above. Some of the l^i's are ε, but the last one is a, and maybe others too are $\neq \varepsilon$. Hence this path p in G has the form

$(\stackrel{\varepsilon^*}{\to}\stackrel{l}{\to})^*$, where l stands for labels $\neq \varepsilon$, and the last label is a from above. Since G, G' are bisimilar we can replicate p in G' with a path p', having the same labels, and assuring that after each label we end in a node which is bisimilar with the corresponding one in G. Based on p' we can replicate our entire path $p_1 = x_1 \stackrel{\varepsilon^*}{\to}\stackrel{b}{\to} y$ in $e(G')$ as follows. Every ε edge in p' becomes a "simple" tree consisting of a single ε edge in $e(G')$, and we traverse that one. Every edge $w' \stackrel{l}{\to} z'$ in p' with $l \neq \varepsilon$ has a corresponding edge $w \stackrel{l}{\to} z$ in p, with z, z' bisimilar. Hence, by our induction hypothesis the trees $T_{w \stackrel{l}{\to} z}$ and $T_{w' \stackrel{l}{\to} z'}$ are bisimilar too, and R_1 contains also a bisimilarity relation between them. Except for the last label l, the path p_1 traverses the tree $T_{w \stackrel{l}{\to} z}$ only through ε edges, all the way from the root to a leaf labeled X: we have taken care in our definition of bisimilarity (Definition 2.1) to ensure that in this case there is a corresponding ε path from the root of $T_{w' \stackrel{l}{\to} z'}$ to some leaf labeled X, and we follow this path in $e(G')$. For the last label $l = a$, we also have the trees $T_{u \stackrel{a}{\to} v}$ and $T_{u' \stackrel{a}{\to} v'}$ bisimilar. The last portion of the path p_1 is now a path of the form $\stackrel{\varepsilon^*}{\to}\stackrel{b}{\to}$ from the root of $T_{u \stackrel{a}{\to} v}$ to y, which we can replicate in $T_{w' \stackrel{l}{\to} z'}$ and end in some y' which is bisimilar to y.

□

6.3 UnCAL Semantics on Notation of Trees

Recall our syntax for trees given in Section 2:

$$\begin{array}{rcl} T & ::= & \{\} \mid \{l \Rightarrow T\} \mid T_1 \cup T_2 \mid X \mid \\ & & (T \text{ where } X_1 := T_1, \ldots, X_n := T_n) \end{array}$$

UnCAL admits a simple semantics in terms of data represented like that. Namely the semantics of $\{\}, \{l \Rightarrow t\}, t_1 \cup t_2$ is the obvious one. $t \mathbin{@}_{\mathcal{X}} \vec{s}$ is "marker substitution": in the tree t we substitute each marker X_i, $i = 1, k$, with the corresponding tree from $\vec{s}$. As in any formalism with free and bound variables, we have to do bounded marker renaming in t before the substitution, in order to avoid unwanted capturings of the "free" markers in $\vec{s}$. E.g. let $t = (Y \text{ where } Y = \{a \Rightarrow X, b \Rightarrow Y\})$ and s be the $\{X\}$-indexed family $X := \{c \Rightarrow Y\}$. Here t has a single output marker X: Y is used in t only to define a cycle. But s *does* have the marker Y. To avoid capturing this Y by t, we first rename Y in t to Z: $t = (Z \text{ where } Z = \{a \Rightarrow X, b \Rightarrow Z\})$, then $t \mathbin{@}_{\{X\}} s = (Z \text{ where } Z = \{a \Rightarrow \{c \Rightarrow Y\}, b \Rightarrow Z\})$.

Finally we define *gext*. On tree notations it is best to visualize *gext* as a *recursive process*. Let $g(T) = gext_{\{X\}}(f)(T)$, and assume first that $f(l, t)$ does not depend[5] on the tree variable t, i.e. f is of the form $f(l)$. Then:

$$\begin{array}{rcl} g(\{\}) & \stackrel{\text{def}}{=} & \{\} \\ g(\{l \Rightarrow t\}) & \stackrel{\text{def}}{=} & f(l) \mathbin{@}_{\{X\}} g(t) \\ g(T_1 \cup T_2) & \stackrel{\text{def}}{=} & g(T_1) \cup g(T_2) \\ g(Y) & \stackrel{\text{def}}{=} & X \cdot Y \quad Y \text{ a marker} \\ g(T \text{ where } Y_1 = T_1, \ldots, Y_n = T_n) & \stackrel{\text{def}}{=} & g(T) \text{ where } X \cdot Y_1 = g(T_1), \ldots, X \cdot Y_n = g(T_n) \end{array}$$

This suggests a computation method in which we start from the root and go downwards in the tree. The last two equations explain how we deal with cycles.

In the more general case, $f(l, t)$ really uses t: here we need to complicate a bit the semantics of *gext*, in order to handle well cyclic structures. To see the problem, consider the following function $g(T) = gext_{\{X\}}(f)(T)$, with $f(l, t) = \text{if } isempty(t) \text{ then } \{\} \text{ else } \{l\} \cup X$, which computes the set of all labels leading to a non-leaf node. Consider now the tree $T = Y_1 \text{ where } (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\})$, on which g should return $\{a, b\}$. Our previous definition would give us:

$$\begin{array}{rl} g(Y_1 \text{ where } (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\})) = & \\ = & g(Y_1) \text{ where } (X \cdot Y_1 = g(\{a \Rightarrow Y_2\}), X \cdot Y_2 = g(\{b \Rightarrow Y_1\})) \\ = & X \cdot Y_1 \text{ where } (X \cdot Y_1 = f(a, Y_2) \mathbin{@} g(Y_2), X \cdot Y_2 = f(b, Y_1) \mathbin{@} g(Y_1)) \end{array}$$

[5] Hence $gext_{\{X\}}(f)(T)$ is in fact $vext_{\{X\}}(f)(T)$, see Subsection 6.4.

This is certainly not what we want, because in f we will end up testing $isempty(Y_1)$ and $isempty(Y_2)$ (which are both true). Instead, we have to do one substitution step before computing the function f.

For this, we define an *environment* to be a sequence of marker,tree pairs, like $Env = [X_1 = T_1, \ldots, X_n = T_n]$. There may be multiple occurrences of the same marker X_i, i.e. $X_i = X_j$ for $i < j$: in this case X_i overrides X_j. Given some tree T, $T[Env]$ denotes the result of simultaneously substituting each marker X_i with T_i in T. We define now the evaluation of $gext_{\{X\}}(f)(T)$ under some environment Env, as follows:

$$
\begin{array}{rcl}
g(\{\}, Env) & \stackrel{\text{def}}{=} & \{\} \\
g(\{l \Rightarrow t\}, Env) & \stackrel{\text{def}}{=} & f(l, t[Env]) \mathbin{@_{\{X\}}} g(t) \\
g(T_1 \cup T_2, Env) & \stackrel{\text{def}}{=} & g(T_1, Env) \cup g(T_2, Env) \\
g(Y, Env) & \stackrel{\text{def}}{=} & X \cdot Y \quad Y \text{ a marker} \\
g(T \;\textsf{where}\; Y_1 = T_1, \ldots, Y_n = T_n, Env) & \stackrel{\text{def}}{=} & g(T, Env') \;\textsf{where}\; X \cdot Y_1 = g(T_1, Env'), \ldots, X \cdot Y_n = g(T_n, Env') \\
 & & \text{where } Env' = [Env'', Env] \\
 & & \text{and } Env'' = [Y_1 = (Y_1 \;\textsf{where}\; Y_1 = T_1, \ldots, Y_n = T_n), \ldots, \\
 & & \quad Y_n = (Y_n \;\textsf{where}\; Y_1 = T_1, \ldots, Y_n = T_n)]
\end{array}
$$

Note that the recursive function, g, is always applied to simpler arguments, hence the recursive process will terminate. But when we compute $f(l, t)$, we replace t with $t[Env]$, thus compensating for the loss of information which occurred when g was computed on a where construct. For example, consider the revised computation for the example above, and let $Env = [Y_1 = (Y_1 \;\textsf{where}\; (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\})), Y_2 = (Y_2 \;\textsf{where}\; (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\}))]$. Then:

$$
\begin{array}{rl}
\multicolumn{2}{l}{g(Y_1 \;\textsf{where}\; (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\}), []) =} \\
= & g(Y_1, Env) \;\textsf{where}\; (X \cdot Y_1 = g(\{a \Rightarrow Y_2\}, Env), X \cdot Y_2 = g(\{b \Rightarrow Y_1\}, Env)) \\
= & X \cdot Y_1 \;\textsf{where}\; (X \cdot Y_1 = f(a, Y_2[Env]) \mathbin{@} g(Y_2, Env), X \cdot Y_2 = f(b, Y_1[Env]) \mathbin{@} g(Y_1, Env)) \\
= & X \cdot Y_1 \;\textsf{where}\; (X \cdot Y_1 = f(a, Y_2 \;\textsf{where}\; (\ldots)) \mathbin{@_{\{X\}}} X \cdot Y_2, \\
 & \qquad X \cdot Y_2 = f(b, Y_1 \;\textsf{where}\; (\ldots)) \mathbin{@_{\{X\}}} X \cdot Y_1) \\
= & X \cdot Y_1 \;\textsf{where}\; (X \cdot Y_1 = ((\{a\} \cup X) \mathbin{@_X} X \cdot Y_2), X \cdot Y_2 = ((\{b\} \cup X) \mathbin{@_X} X \cdot Y_1)) \\
= & X \cdot Y_1 \;\textsf{where}\; (X \cdot Y_1 = (\{a\} \cup X \cdot Y_2), X \cdot Y_2 = (\{b\} \cup X \cdot Y_1))
\end{array}
$$

The last result, also cumbersome, is equivalent (via translation into graphs, and then bisimulation) to $\{a, b\}$. Note that the computation of $f(a, Y_2 \;\textsf{where}\; (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\}))$ requires the computation of $isempty(Y_2 \;\textsf{where}\; (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\}))$. The semantics of *isempty* when applied to where-constructs involves the computation of transitive closure, as in the case of graphs: in our particular example, $isempty(Y_2 \;\textsf{where}\; (Y_1 = \{a \Rightarrow Y_2\}, Y_2 = \{b \Rightarrow Y_1\})) = false$.

Now we turn to the general case, when $\mathcal{X} = \{X_1, \ldots, X_k\}$. Then $gext_{\mathcal{X}}$ allows us to define simultaneously k mutually recursive functions g_i, $i = 1, k$, as shown in Figure 13 where, for illustration, we assume that $f(l, t)$ does not depend on t. Unless otherwise specified, we take $gext_{\mathcal{X}}(f)(T)$ to be $g_1(T)$. Note that when we apply $gext_{\mathcal{X}}(f)$ to a tree with n markers $Y_1, \ldots, Y_n$, we get a tree with kn markers, labeled $X_i \cdot Y_j$, $i = 1, k$, $j = 1, n$. When $f(l, t)$ does depend on t, then we use environments, as above.

At this point we have defined the semantics of UnCAL queries on notations of trees. As in the case of rooted, labeled trees, there are different tree notations for the same data, but we cannot give a proposition similar to Proposition 6.1, because it is cumbersome to define when two tree notations denote the same data (for tree notations we do not have a concept similar to that of bisimilarity on rooted, labeled graphs). Instead, we can prove that the two semantics for UnCAL, on trees and on rooted, labeled graphs, are the same:

Proposition 6.2 *Let e be an UnCAL query, T be a tree notation, and G be it's associated rooted, labeled graph. Let $T' = e(T)$ (i.e. e's semantic on tree notations, according to this subsection), and $G' = e(G)$ (i.e. e's semantic on graphs, according to Subsection 6.3). Then G' is bisimilar to the the rooted, labeled graph associated to T'.*

$$
\begin{array}{rclcrcl}
g_1(\{\}) & \stackrel{\text{def}}{=} & \{\} & \ldots & g_k(\{\}) & \stackrel{\text{def}}{=} & \{\} \\
g_1(\{l \Rightarrow t\}) & \stackrel{\text{def}}{=} & f_1(l) \ @_{\mathcal{X}}\ g(t) & \ldots & g_k(\{l \Rightarrow t\}) & \stackrel{\text{def}}{=} & f_k(l) \ @_{\mathcal{X}}\ g(t) \\
g_1(T_1 \cup T_2) & \stackrel{\text{def}}{=} & g_1(T_1) \cup g_1(T_2) & \ldots & g_k(T_1 \cup T_2) & \stackrel{\text{def}}{=} & g_k(T_1) \cup g_k(T_2) \\
g_1(Y) & \stackrel{\text{def}}{=} & X_1 \cdot Y & \ldots & g_k(Y) & \stackrel{\text{def}}{=} & X_m \cdot Y
\end{array} \quad (1)
$$

$$
\begin{array}{ll}
g_1(T \ \mathsf{where}\ Y_1 := T_1, \ldots, Y_n := T_n) \stackrel{\text{def}}{=} & g_k(T \ \mathsf{where}\ Y_1 := T_1, \ldots, Y_n := T_n) \stackrel{\text{def}}{=} \\
\stackrel{\text{def}}{=} g_1(T) \ \mathsf{where}\ (X_i \cdot Y_j := g_i(T_j), & \stackrel{\text{def}}{=} g_k(T) \ \mathsf{where}\ (X_i \cdot Y_j := g_i(T_j), \\
\quad i = 1, k, j = 1, n) & \quad i = 1, k, j = 1, n)
\end{array}
$$

Figure 13: The definition of $gext_{\mathcal{X}}(f)(T)$, when $f(l,t)$ does not depend on t. Here $f(l)$ and $g(t)$ are $\mathcal{X}$-indexed families $\{X_1 := f_1(l), \ldots, X_k := f_k(l)\}$, $\{X_1 := g_1(t), \ldots, X_k := g_k(t)\}$ and $g(T) = gext_{\mathcal{X}}(f)(T)$.

Proof: (Sketch) The proof is done by induction on the structure of e. The only nontrivial case is when e is a *gext* expression. For illustration, we take a particular case of a *gext* expression, say $gext_{\{X\}}(f)(T)$, where T is the input tree. Let g denote the function $gext_{\{X\}}(f)$. Then $g(T)$ is computed inductively, based on the structure of the sytnax of the tree T, starting with the empty environment, that is $g(T, [])$. As we apply g to tree subexpressions T', the environment will grow: $g(T', Env)$. We prove by induction on the structure of the subexpressions T' that: (1) the graph representation G' of T' is a subgraph of the graph representation G of T, thus every edge $v \xrightarrow{a} w$ in G' is in G too; and (2) for each edge $v \xrightarrow{a} w$ of G', the graph rooted at w in the graph representation of $T'[Env]$ is bisimilar with the graph rooted at w in G; here we note that G' is a subgraph of the graph representation of $T'[Env]$, so that the latter contains the edge $v \xrightarrow{a} w$ from G'. Condition (1) ensures that during the recursive process evaluating g on the subexpressions T', the function f is applied in the same manner as in the parallel process, which applies f in parallel to all edges in G. Condition (2) ensures that, whenever f is applied in the recursive process to a and t, $f(a,t)$, the tree t is the "same" as the corresponding tree in the application of f during the parallel process. Details omitted. □

6.4 Particular Instances of *gext*

We specialize $gext_{\mathcal{X}}(f)$ to two important cases. The first is when $\mathcal{X} = \emptyset$.[6] In this case only the topmost level of the tree denoted by t is processed. This is a classical operation on sets which we call *ext*:

$$\frac{f : Label \times Tree \rightarrow Tree \quad t : Tree}{ext(f)(t) : Tree}$$

with the meaning $ext(f)(\{l_1 \Rightarrow t_1, \ldots, l_n \Rightarrow t_n\}) = f(l_1, t_1) \cup \cdots \cup f(l_n, t_n)$. The second case is when $f(l,t)$ depends only on the label l, not on the tree t. We call this *vext*.

$$\frac{f : Label \rightarrow Tree^{\mathcal{X}}_{\mathcal{X}} \quad t : Tree_{\mathcal{Y}}}{vext_{\mathcal{X}}(f)(t) : Tree^{\mathcal{X}}_{\mathcal{X}\cdot\mathcal{Y}}}$$

When $\mathcal{X}$ has a single marker X, then *vext* is exactly *ext* on lists. Indeed, assume that we encode a list $[A_1, \ldots, A_n]$ "vertically" as $\{A_1 \Rightarrow \{A_2 \Rightarrow \ldots \{A_n\} \ldots\}\}$, and assume that $f(l)$ returns a list "ending in X" (i.e., of the form $\{B_1 \Rightarrow \{B_2 \Rightarrow \ldots \{B_m \Rightarrow X\} \ldots\}\}$), for every label l. Then *vext* is described by $vext(f)([A_1, \ldots, A_n]) \stackrel{\text{def}}{=} f(A_1) \ @ \ldots @ \ f(A_n)$.

An example of *vext* is the query $deepflatten(T)$, which returns the set of all subtrees of t. In UnQL, this is expressed as

select t

where $\_* \Rightarrow \backslash t \leftarrow T$

In UnCAL we express it as the X_1-component of:

$vext_{\{X_1,X_2\}}\ (\lambda l.X_1 := X_1 \cup \{l \Rightarrow X_2\}, X_2 := \{l \Rightarrow X_2\})(T)$

More generally, for any deterministic finite automaton there exists a $vext_{\mathcal{X}}$ expression that pulls up to the top level all subtrees reachable by a path accepted by that automaton. Moreover, the automaton may have not only

[6] By convention, in this case f returns a single tree, and $gext_{\mathcal{X}}(f)$ returns a single tree too.

labels, but also previously bound variables, or even negation (i.e., $\neq l$) on its edges. The number of markers in $\mathcal{X}$ is one plus the number of states of the automaton. In particular, all queries select t where $R \Rightarrow \backslash t \leftarrow T$ from Section 3, where R is a regular expression, can be translated into *vext*.

6.5 Equivalence of UnQL and UnCAL, and Applications

We prove here that UnQL and UnCAL are equivalent. As a consequence we will be able to prove properties about UnQL by proving them about UnCAL, which is easier to deal with. Namely we will prove that all UnQL queries are in PTIME, and that UnQL queries over trees representing flat relations are equivalent to the relational algebra (Theorem 5.1).

Theorem 6.3 *UnQL and UnCAL express the same set of queries. Moreover, there exists an effective translation from UnQL to UnCAL.*

Proof: (Sketch) One direction is easy. Every UnCAL expression can obviously be translated into UnQL (*gext* becomes a traverse). The other direction is more involved and we sketch it briefly next.

Consider first a select e where $C_1, C_2, \ldots, C_m$ construct. Without loss of generality, we may assume that it is in a canonical form, in which the selectors $C_1, C_2, \ldots$ are in one of the following forms: (1) $\backslash l \Rightarrow \backslash t \leftarrow T$, (2) $R \Rightarrow \backslash t \leftarrow T$ where R is a regular expression, or (3) a predicate p. It is easy to transform every select query into such a canonical form, e.g., by replacing $\backslash l_1 \Rightarrow \backslash l_2 \Rightarrow \backslash t \leftarrow T$ with $\backslash l_1 \Rightarrow \backslash t' \leftarrow T, \backslash l_2 \Rightarrow \backslash t \leftarrow t'$, etc. Then we translate select e where $C_1, C_2, \ldots, C_m$ into an UnCAL expression as follows. If C_1 is of the form $\backslash l \Rightarrow \backslash t \leftarrow T$, the translation is $ext\,(\lambda(l,t).(\text{select } e \text{ where } C_2, \ldots, C_m))\,(T)$. If C_1 is of the form $R \Rightarrow \backslash t$, we translate it into $ext\,(\lambda(l,t).(\text{select } e \text{ where } C_2, \ldots, C_m))\,(regex_R(T))$, where $regex_R$ is the UnCAL epxressions retrieving all subtrees accessible by the regular path expression R. Finally, when C_1 is a predicate p, the translation is if p then select e where $C_2, \ldots, C_m$ else $\{\}$. Of course, in all three cases we continue to translate the remaining select expression. Eventually we reach $m = 0$, when select e where becomes simply e.

Next, consider a traverse expression. Here we first restructure it into a canonical form: traverse T giving $X_1, \ldots, X_k$ case $\backslash l \Rightarrow \backslash t$ then $X_1 := e_1, \ldots, X_k := e_k$: the expressions $e_1, \ldots, e_k$ will include (under if-constructs) all patterns and conditions. It is obvious now that this canonical form is precisely the $gext_{\mathcal{X}}(f)(T)$ construct, where $f(l,t) = \{X_1 := e_1, \ldots, X_k := e_k\}$. □

UnCAL is a simpler formalism than UnQL. Hence we can more easily prove properties about UnCAL than about UnQL. For example:

Theorem 6.4 *All queries expressible in UnCAL (and, hence, in UnQL) are defined on all cyclic structures and are computable in* PTIME.

Finally, we end this subsection with a proof of Theorem 5.1:

Proof: (of Theorem 5.1) Let NRA stand for the *nested relational algebra* [BNTW95]. A quick way to introduce it is to notice that it extends the relational algebra (RA) with the operations nest and unnest ***** paredaens92 *****. It is easy to show that UnQL can express all RA operators. unnest (T), where T encodes a nested relation, say of type $\{[A : String, B : \{[C : String]\}]\}$ can be expressed in UnQL as:

select $\{Tup \Rightarrow \{A \Rightarrow s, B \Rightarrow t\}\}$
where $Tup \Rightarrow \{A \Rightarrow \backslash s, B \Rightarrow Tup \Rightarrow C \Rightarrow \backslash t\} \leftarrow T$

nest (T) can be expressed as follows, where T is a flat relation of type $\{[A : String, B : String]\}$ and the result has type $\{[A : String, B : \{[C : String]\}]\}$:

select $\{Tup \Rightarrow \{\ A \Rightarrow s \Rightarrow \{\},$
$\quad B \Rightarrow$ select $\{Tup \Rightarrow \{C \Rightarrow t\}\}$
$\quad\quad$ where $Tup \Rightarrow \{A \Rightarrow s \Rightarrow \_, B \Rightarrow \backslash t\} \leftarrow T\}\}$
where $Tup \Rightarrow A \Rightarrow \backslash s \Rightarrow \_ \in DB$

The converse is a little more involved. For the rest of the proof we will view queries as *functions*, i.e. an UnQL query e is a function of its input tree $e(T)$. The converse relies on two facts:

Bounded depth For any UnQL function e there exists a polynomial P such that whenever T is a (cycle-free) tree of depth d, then $e(T)$ is also cycle free and has depth $\leq P(d)$.

Encoding of UnQL into NRA For every depth d there exists an encoding of UnQL trees of depts $\leq d$ as nested relations of some type τ_d, such that for every UnQL function e there exists an NRA function e_d which "simulates" e on trees of depth $\leq d$.

The "bounded depth" property is easy to prove: formally we do it by induction of the structure of UnCAL functions. For the encoding, we define $\tau_{d+1} \stackrel{\text{def}}{=} \{[A : Label \cup \{\varepsilon\}, B : \tau_d]\}$, and $\tau_0 \stackrel{\text{def}}{=} unit$ ($unit$ is the type consisting only of the empty record $[]$). We treat $Label_\varepsilon = Label \cup \{\varepsilon\}$ as a variant type and copile it away, as in [Won94]. E.g.

$$\tau_3 = \{[A : Label_\varepsilon, B : \{[A : Label_\varepsilon, B : \{[A : Label_\varepsilon, B : unit]\}]\}]\}$$

Next, we encode every tree $t = \{l_1 \Rightarrow t_1, \ldots, l_n \Rightarrow t_n\}$ of depth $\leq d$ as a nested relation of type τ_d, namely as $\{[A = l_1, B = r_1], \ldots, [A = l_n, B = r_n]\}$, where the nested relations $r_1, \ldots, r_n$ encode the trees $t_1, \ldots, t_n$ respectively. For example the tree $\{a, b \Rightarrow \{c\}\}$ is encoded in τ_3 as:

$$\{[A = a, B = \{\}], [A = b, B = \{[A = c, B = \{\}]\}]\}$$

More, we can extend the encoding τ_d to encode trees with markers: a tree with markers $X_1, \ldots, X_k$ and depth $\leq d$ will be encoded by a nested relation of type

$$\tau_{d,\{X_1,\ldots,X_k\}} \stackrel{\text{def}}{=} \{[A = Label_\varepsilon, B = \tau_{d-1}]\} + \underbrace{unit + \ldots + unit}_{k \text{ times}}$$

where $+$ denotes disjoint type union, and can be compiled away in NRA [Won94]. Next we construct for every UnCAL function e and every number d, a "translated" NRA function $e_d : \tau_d \to \tau_{P(d)}$ (where P is the polynomial associated to e; more precisely, the type of e_d is $\tau_{d,\mathcal{X}} \to \tau_{P(d),\mathcal{Y}}$, if the input of e is in $Tree_\mathcal{X}$ and its output in $Tree_\mathcal{Y}$), with the following meaning: whenever T is a tree of depth $\leq d$ and $R : \tau_d$ encodes T, then $f(R)$ encodes $e(T)$. We sketch this for three cases of increasing difficulty. In general, we need to consider functions e with more than one inputs, say $e(T_1, T_2, \ldots, T_p)$.

Union Say $e(T) = e'(T) \cup e''(T)$. Translate e', e'' first, to get $e'_d : \tau_d \to \tau_{P'(d)}$ and $e''_d : \tau_d \to \tau_{P''(d)}$. Assume $P'(d) \geq P''(d)$. Then the translation of e is $e_d(R) : \tau_d \to \tau_{P'(d)}$, $e_d(R) \stackrel{\text{def}}{=} e'_d(R) \cup i(e''_d(R))$, where $i : \tau_{P''(d)} \to \tau_{P'(d)}$ is the "injection", leaving the input unchanged, but lifting the types of the empty sets on the leaves.

@ For illustration we consider the simple case $T_1 \ @_{\{X\}} \ T_2$, i.e. when append is done on a single marker. For any two numbers d_1, d_2 we translate it into $append_{d_1,d_2} : \tau_{d_1,\{X\}} \times \tau_{d_2} \to \tau_{d_1+d_2}$: we do this by a straightforward induction on d_1.

gext We consider the simple case of a single marker, say $e(T) = gext(f)_{\{X\}}(T)$; recall that f has two parameters, a label l and a tree t, $f(l, t)$. Then note that on some tree $T = \{a_1 \Rightarrow T_1, \ldots, a_n \Rightarrow T_n\}$, $e(T)$ returns $(f(a_1, T_1) \ @ \ e(T_1)) \cup \ldots \cup (f(a_n, T_n) \ @ \ e(T_n))$. Here we use the fact that NRA is closed under ext, i.e. whenever $h : \tau \to \{\tau'\}$ is definable in NRA, then $ext(h) : \{\tau\} \to \{\tau'\}$ defined by $ext(h)(\{x_1, \ldots, x_n\}) = h(x_1) \cup \ldots \cup h(x_n)$ is also in NRA: we will construct e_d with the aid of an ext expression. First translate f into $f_d : Label \times \tau_d \to \tau_{P(d),\mathcal{X}}$. Then we translate the whole expression e, but for $d-1$ instead of d, to get $e_{d-1} : \tau_{d-1} \to \tau_{Q(d-1)}$, where Q is the polynomial associated to e, to be shown next. We can do this, because we construct the translation e_d both on induction of the structure of e *and* on the number d. Now define $e_d(R) \stackrel{\text{def}}{=} ext(\lambda(l, r).append_{P(d),Q(d-1)}(f_d(l, r), e_{d-1}(R)))(R)$. Finally, note that $Q(d) = P(d) + Q(d-1) = P(d) + P(d-1) + P(d-2) \ldots \leq dP(d)$.

Now we can finally prove that every UnCAL query e on trees encoding nested relations is equivalent to an NRA query. Let f be an UnCAL query which maps encodings of (nested) relational databases to encodings of (nested) relational databases. More precisely, there exists two nested relation types τ, τ' such that for any tree T encoding a nested relational database of type τ (as in Figure 2, but generalized to nested relations), $e(T)$ is a tree encoding, in the same way, of a nested relation of type τ'. We will show that f is expressible in NRA. Let d, d' be the depths of trees encoding τ and τ' respectively ($d = d' = 4$ for relational databases).

Then these trees can be encoded back as nested relation types $\tau_d, \tau_{d'}$ (which are more complex than τ, τ'). Our key observation is that the functions $encode : \tau \rightarrow \tau_d$ and $decode : \tau_{d'} \rightarrow \tau'$, which we describe in a moment, are expressible in NRA. Then the translation of e into NRA is simply the function $f : \tau \rightarrow \tau'$, $f \stackrel{\text{def}}{=} decode \circ f_d \circ encode$. Moreover, when τ, τ' are flat relation types, then using the conservativity extension property in [PG92, Won93], one can prove that f is expressible in RA.

So it remains to show that $encode, decode$ are expressible in NRA. Start with $encode : \tau \rightarrow \tau_d$. For illustration, suppose $\tau = \{[M : String, N : String]\}$ is the type of a binary relation. Simplifying the example in Figure 2, we can represent a relation $R = \{[M = x_1, N = y_1], \ldots, [M = x_n, N = y_n]\}$ as a tree of depth 3, $T = \{tup \Rightarrow \{M \Rightarrow \{x_1\}, N \Rightarrow \{y_1\}\}, \ldots, tup \Rightarrow \{M \Rightarrow \{x_n\}, N \Rightarrow \{y_n\}\}\}$. T itself is further encoded as a nested relation of type τ_3, see above. Basically $encode(R)$ has to return $\{[A = tup, B = \{[A = M, B = \{[A = x_1, B = \{\}]\}], [A = N, B = \{[A = y_1, B = \{\}]\}]\}], \ldots\}$. It is tedious, but straightforward to express $encode$ in NRA: note that $decode$ uses the label tup as a constant. Conversely, $decode$ has to take an encoding (of type $\tau_{d'}$) and produce the originating (nested) relation of type τ'. This is done similarly, with the observation that $decode$ relies on the input to be a correct encoding: else, the result of $decode$ is unspecified.

$\square$

6.6 Optimizations in UnCAL

The main advantage of having a simple formalism is that it enables us to derive optimization rules. Optimization rules for set-based query languages centered around *ext* and set union $\cup$ have been considered previously [Won94]. Surprisingly, they hold in identical form for the vertical dimension, with the slight twist that sometimes we have to restrict ourselves to *vext* instead of *gext*. Rather than considering all optimization rules, we shall focus on two: the first is a simple equation pushing selections down a join tree, the second a rather complex loop fusion, putting maximum strain on the usage of markers in UnCAL.

Example 6.5 We start with the following simple optimization rule for *ext*, adapted from [Won94]:

$ext\ (\lambda(l,t).\text{if } p \text{ then } e \text{ else } e')\ (T) \ =$

$\quad \text{if } p \text{ then } ext\ (\lambda(l,t).e)\ (T) \text{ else } ext\ (\lambda(l,t).e')\ (T)$

(when l and t do not occur free in p)

This extends naturally to *vext* and *gext* as:

$gext\ (\lambda(l,t).\text{if } p \text{ then } e \text{ else } e')\ (T) \ =$

$\quad \text{if } p \text{ then } gext\ (\lambda(l,t).e)\ (T) \text{ else } gext\ (\lambda(l,t).e')\ (T)$

(when l and t do not occur free in p)

Using this, we derive the classical optimization rule pushing selections down a join tree for the vertical dimension of UnCAL. To see how this works, consider the following *vertical selection-join* query:

traverse t_1 giving X
  case $\backslash a \Rightarrow$ _ then
    $X :=$ (traverse t_2 giving Y
        case $\backslash b \Rightarrow$ _ where $p(a)$ then
          $Y := \{f(a,b) \Rightarrow Y\}$)

Here each edge a of t_1 satisfying $p(a)$ is replaced with a full copy of the tree t_2; edges not satisfying $p(a)$ are removed. The copy of t_2 is obtained by replacing each of its labels b with a new label $f(a,b)$. To make the query nontrivial, we assume that t_2 had initially some markers X. The query gets translated into the following UnCAL expression: $vext_{\{X\}}\ (\lambda a.\ vext_{\{Y\}}\ (\lambda b.\text{if } p(a) \text{ then } \{f(a,b) \Rightarrow Y\} \text{ else } \{\})\ (t_2))\ (t_1)$.

The problem is that the predicate p is applied in the inner loop, while it would be obviously less expensive to apply it in the outer loop, as in:

traverse t_1 giving X
  case $\backslash a \Rightarrow$ _ where $p(a)$ then
    $X :=$ (traverse t_2 giving Y

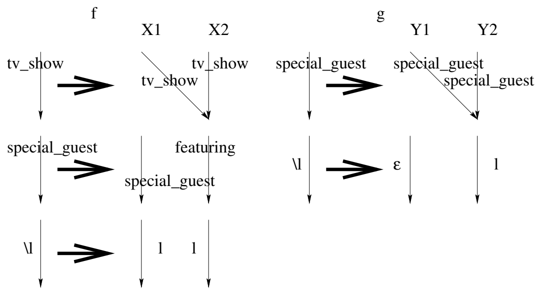


Figure 14: Graphic representation of the functions f and g.

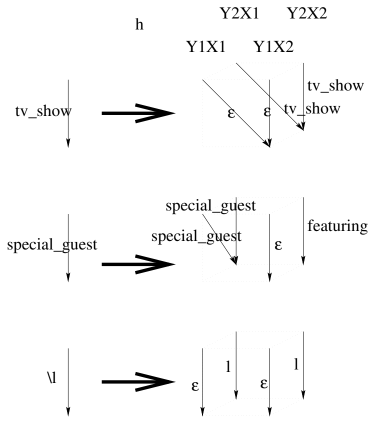


Figure 15: Graphic representation of the function $h = vext(g) \circ f$.

case $\backslash b \Rightarrow$ _ then
  $Y := \{f(a, b) \Rightarrow Y\})$

which is translated into $vext_{\{X\}}$ (λa.if $p(a)$ then $vext_{\{Y\}}$ $(\lambda b.\{f(a, b) \Rightarrow Y\})$ (t_2) else $\{\}$) (t_1). The rule for if described earlier and the simple rule $vext(f)(\{\}) = \{\}$ allows us to transform the first query into the second one.

The second optimization rule corresponds to the *vertical loop fusion* in [GP84]. Consider the expression $vext(g)(vext(f)(t))$. One can visualize it as a pipeline process in which the edges of t are fed one by one into the function f, which transforms every label into a "small" tree $f(a)$. The new edges b produced by f are further fed into g, which replaces each such edge with another "small" tree $g(b)$. One can optimize this process by traversing t only once, and replacing every edge a with the tree $vext(g)(f(a))$. This makes the intermediate result $vext(f)(t)$ unnecessary, and may even speed up the computation dramatically (e.g., when some of the $g(b)$ trees are empty) by pruning early the computation of $vext(f)$. In short, we perform an optimization by replacing $vext(g)(vext(f)(t))$ with $vext(vext(g) \circ f)(t)$ (here $\circ$ denotes function composition).

Getting this rule to work well was difficult until we discovered the way in which $vext_{\mathcal{X}}(f)(t)$ must interact with markers existing in t, which is essentially given in equation (1). Before listing the optimization rules, let us consider an example.

Example 6.6 Take the query in Example 4.2, which replaces all *SpecialGuest* edges with *Featuring* edges in the *TVShows* of the movie database, and suppose this query defines a view T. T can be expressed as $vext_{\{X_1, X_2\}}(f)(DB)$, where f is graphically represented in Figure 14. Now assume that we want to query the view to obtain the set of all *SpecialGuests* in T. This can be written as select $\{SpecialGuest \Rightarrow t\}$ where $\_* \Rightarrow SpecialGuest \Rightarrow \backslash t \in T$ or, equivalently, as:

traverse T giving Y_1, Y_2
  case $SpecialGuest \Rightarrow$ _ then

$Y_1 := \{SpecialGuest \Rightarrow Y_2\}$

case $\backslash a \Rightarrow$ _ then

$Y_1 := Y_1, Y_2 := \{a \Rightarrow Y_2\}$

This is precisely $vext_{\{Y_1,Y_2\}}(g)(T)$, where the function g is represented graphically in Figure 14. Querying the view T entails computing the composition $vext_{\{Y_1,Y_2\}}(g)(vext_{\{X_1,X_2\}}(f)(DB))$. This is a pipeline process as described earlier, and is equivalent to $vext_{\mathcal{Z}}(h)(DB)$, where $\mathcal{Z} = \mathcal{Y} \cdot \mathcal{X} = \{Y_1 \cdot X_1, Y_2 \cdot X_1, Y_1 \cdot X_2, Y_2 \cdot X_2\}$, and h is obtained by applying g locally on the "small" trees produced by f. The best way to understand the function h is graphically (see Figure 15). Nevertheless, we can write down the resulting query, consisting of a single traverse expression:

traverse T giving $Y_1 \cdot X_1, Y_2 \cdot X_1, Y_1 \cdot X_2, Y_2 \cdot X_2$

case $TVShow \Rightarrow$ _ then

$Y_1 \cdot X_1 := Y_1 \cdot X_2,$

$Y_2 \cdot X_1 := \{TVShow \Rightarrow Y_2 \cdot X_2\})$

case $SpecialGuest \Rightarrow$ _ then

$Y_1 \cdot X_1 := \{SpecialGuest \Rightarrow Y_2 \cdot X_1\},$

$Y_2 \cdot X_2 := \{Featuring \Rightarrow Y_2 \cdot X_2\})$

case $\backslash l \Rightarrow$ _ then

$Y_1 \cdot X_1 := Y_1 \cdot X_1,$

$Y_1 \cdot X_2 := Y_1 \cdot X_2,$

$Y_2 \cdot X_1 := \{l \Rightarrow Y_2 \cdot X_1\},$

$Y_2 \cdot X_2 := \{l \Rightarrow Y_2 \cdot X_2\}$

Further optimizations are possible by "reducing" the number of states (markers): in our example the tree corresponding to $Y_1 \cdot X_2$ is empty, because the only edge leaving from this "state" is an ϵ-edge into the same state, hence this marker can be eliminated.

We give next the optimization rules in their general form. As mentioned earlier, all 9 rules in [Won94] apply to the vertical operators in UnCAL. Instead of listing all of them, we show only the most powerful ones, together with their corresponding horizontal versions.

Theorem 6.7 *The following equations hold:*

$ext(f)(t_1 \cup t_2) = ext(f)(t_1) \cup ext(f)(t_2)$

$vext_{\mathcal{X}}(f)(t_1 \ @_{\mathcal{Y}} \ t_2) = vext_{\mathcal{X}}(f)(t_1) \ @_{\mathcal{X}\cdot\mathcal{Y}} \ vext_{\mathcal{X}}(f)(t_2)$

$ext(g)(ext(f)(t)) = ext(ext(g) \circ f)(t)$

$vext_{\mathcal{X}}(g)(gext_{\mathcal{Y}}(f)(t)) = gext_{\mathcal{X}\cdot\mathcal{Y}}(vext_{\mathcal{X}}(g) \circ f)(t)$

The second equation emphasizes the parallel nature of $vext$. We can compute $vext_{\mathcal{X}}(f)$ on $t_1 \ @_{\mathcal{Y}} \ t_2$ by applying it independently on t_1 and t_2 (note that this fails for $gext$), and then appending the two results. Note how the markers interact here: t_1 has markers $\mathcal{Y}$, while t_2 is a $\mathcal{Y}$-indexed family of trees. Then $vext_{\mathcal{X}}(f)(t_1)$ has markers in $\mathcal{X} \cdot \mathcal{Y}$, hence the index in @ on the right hand side. Since t_2 is a $\mathcal{Y}$-indexed family of trees, $vext_{\mathcal{X}}(f)(t_2)$ is a $\mathcal{X} \cdot \mathcal{Y}$-indexed family of trees, obtained by aggregating all $\mathcal{X}$-indexed families of trees obtained by applying $vext_{\mathcal{X}}(f)$ to each tree in the $\mathcal{Y}$-family t_2.

The last equation is the vertical loop fusion in its most general form: Example 6.6 is an instance of this equation. Note that the loop fusion works even when the first loop is a $gext$: the second loop however needs to be a $vext$. Also note that the new loop function, $vext_{\mathcal{X}}(g) \circ f$, will produce mn markers, where m and n are the number of markers produced by f and g respectively.

7 Conclusions and Related Work

We have adopted a labeled tree abstraction for a wide variety of structured and unstructured data sources, and we have shown that it is possible to develop simple query languages with an underlying optimizable algebra for querying and transforming labeled tree structures. We intend to incorporate at least the comprehension based fragment of UnQL into our existing implementation of CPL (Collection Programming Language [Won94, BDH+95]). However there are some important open questions.

Syntax. The syntax for making deep queries can probably be improved. There are deep restructuring operations expressible in UnCAL whose translation into traverse ... giving ... is rather cumbersome.

Expressive power. Certain restructuring queries appear not to be expressible in UnCAL because they require a fix-point operation. In view of Theorem 5.1 we know that transitive closure (expressed in terms of fixed-depth structures) is not expressible in UnQL. A more interesting problem is a generalization of the "edge merging" example at the end of Section 4. Merging sibling A edges at one level leads to the possibility that more A edges at the next level down will become siblings that can also be merged. It is easy to see that the result of repeatedly merging sibling A edges until no more merges are possible is well-defined, but it does not appear to be possible to compute this in UnQL.

Connections with "graph logic". Another approach to querying labeled trees is to use some variety of "graph logic" [CM90]. It is possible to express at least the *vext* fragment of UnCAL in Datalog, given a graph representation of the input tree. We represent the tree by a set of $(oid, label, oid)$ triples and construct a program that expresses an output graph as a predicate on $(oid', label, oid')$ triples where oid' values are constructed by a Skolem function on the input *oid*s, similar to id-terms in F-logic [KL89]. Whether Datalog optimizations have anything to do with the optimizations we have presented here is an open question.

Other optimizations. Examination of the examples in Section 6 will indicate that, when a schema is present, one can obtain many useful optimizations from "pruning" the search, i.e., not searching subtrees that are known not to contain relevant structures. The problem here is how to describe schemas. We have seen how to do this for relational databases, and the same idea should generalize to object-oriented databases. However, there may be much looser notions of what a schema is for this labeled tree model.

References

[BBW92] 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.

[BDH+95] P. Buneman, S.B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proceedings of 21st VLDB*, September 1995. Also Technical report MS-CIS-95-10, Dept. of Computer and Information Science, University of Pennsylvania. March 1995.

[BDS94] Peter Buneman, Susan Davidson, and Dan Suciu. Programming constructs for unstructured data. Technical Report MS-CIS-95-14, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, March 1994.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of proramming with collection types. *Theoretical Computer Science*, 1995. To appear.

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[CM90] M. P. Consens and A. O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, Nashville, TN, April 1990.

[GP84] A. Goldberg and R. Paige. Stream processing. In *Proceedings of ACM Symposium on LISP and Functional Programming*, pages 53–62, Austin, Texas, August 1984.

[GRS93] N. Goodman, S. Rozen, and L. Stein. Requirements for a deductive query language in the MapBase genome-mapping database. In *Proceedings of Workshop on Programming with Logic Databases, Vancouver, BC*, October 1993.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In M. Stonebraker, editor, *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 393–402, San Diego, California, June 1992.

[KL89] M. Kifer and G. Laussen. F-logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proceedings of ACM-SIGMOD 1989*, pages 46–57, June 1989.

[Kos95] 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.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[PG92] Jan Paredaens and Dirk Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, March 1995.

[TMD92] 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.

[Won93] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993. See also UPenn Technical Report MS-CIS-92-59.

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.