TQL ALGEBRA AND ITS IMPLEMENTATION (EXTENDED ABSTRACT)

Giovanni Conforti, Orlando Ferrara, and Giorgio Ghelli
Università di Pisa
{confor, ferrara, ghelli}@di.unipi.it

Abstract TQL is a query language for semi-structured data. TQL binding mechanism is based upon the *ambient logic*. This binding mechanism is the key feature of TQL, but its implementation is far from obvious, being based on a logic which includes "difficult" operators such as negation, universal quantification, recursion, and new tree-related operators. In [6] an "implementation model" is presented, here we first extend it with tree operations, hence obtaining an algebra for the full TQL language. Then we shortly describe the evaluation techniques that we exploit in the actual implementation.

1. Introduction

TQL is a query language for semi-structured data based on the *tree logic*, an enriched subset of the ambient logic defined in [7, 5]. The *tree logic* is a logic to define sets of trees. It can be naturally used to express types and constraints over semistructured data. As a consequence, problems as subtyping, constraint implication, constraint satisfiability, can all be expressed and investigated as the validity (or satisfiability) of some class of TQL formulae. TQL uses the tree logic as its matching mechanism; as a consequence, more problems, such as query correctness and query containment (and their combinations), become special cases of the validity problem. The high expressivity of the logic allows us to express complex types, constraints, and queries, giving us, for types and constraints, an expressive power that is higher than the one of other proposals [12, 4]. This unified framework for types, constraints, and queries is a central aim of the TQL project, but its further discussion is out of the scope of this paper.

In this paper we describe the foundations of some focal aspects of our implementation of the TQL evaluator. The implementation goes through five steps: *source level rewriting* of the TQL query into a normal form; *translation* of the TQL query into a term of the TQL algebra; *logical optimization* of the algebraic term into a more efficient form; *execution* of the algebraic term. We are still designing a *physical optimization* phase, where queries will be rewritten taking physical information into account. For reasons of space, we cannot go through all those phases, but we will focus on the most original aspects, that are the TQL algebra, and the algorithms to implement its operations.

The major constributions of this paper are: (i) the definition of the TQL algebra, an algebra of operators over trees and over tables (i.e. *relations*) of trees, where both the trees and the tables may be infinite, and the translation of TQL into the TQL algebra; (ii) the description of the implementation of the TQL algebra; the crucial problems we solve are: the finite representation of the infinite tables that arise during evaluation of TQL, and the algorithms used to implement operators such as negation and universal quantification.

2. TQL by examples

Consider the following bibliography, where, informally, $a[F]$ represents a piece of data labelled a with contents F (the data model will be fully defined in the next section); F is empty, or is a collection of similar pieces of data, separated by "|". When F is empty, we can omit the brackets, so that, for example, *Darwen*[ ] can be written as *Darwen*. In this paper we consider a data model where the content F is unordered.

The bibliography below consists of a set of references all labeled *book*. Each entry contains a number of *author* fields, a *title* field, and possibly other fields.

$$
\begin{array}{l}
BOOKS = \ book[\ author[Date] \mid title[DB] \mid publisher[Addison\text{-}Wesley]\ ] \mid \\
\quad book[\ author[Date] \mid author[Darwen] \mid title[Foundation\ for\ Future\ DB] \\
\qquad\quad \mid year[2000] \mid pages[608]\ ] \mid \\
\quad book[\ author[Abiteboul] \mid author[Hull] \mid author[Vianu] \\
\qquad\quad \mid title[Foundation\ of\ DB] \mid publisher[Addison\text{-}Wesley] \mid year[1994]\ ]
\end{array}
$$

Suppose we want to find all the books in *BOOKS* where one author is *Date*; then we can write the following query (hereafter $\mathcal{X}$ and x are variables and everything else is a constant; in the concrete syntax, variable names begin with a $ character):

$$from \quad BOOKS \vDash .book[\mathcal{X}], \quad \mathcal{X} \vDash .author[Date] \quad select \quad text[\mathcal{X}]$$

The query consists of a list of *matching expressions* contained between *from* and *select*, and a *reconstruction expression*, following *select*. The matching expressions bind $\mathcal{X}$ with every piece of data that is reachable from the root *BOOKS* through a *book* path, and such that a path *author* goes from $\mathcal{X}$ to *Date*; the answer is $text[author[Date] \mid title[DB] \mid \ldots] \mid text[author[Date] \mid author[Darwen] \mid \ldots]$, i.e. the first two books in the database, with the outer *book* rewritten as *text*. The operator $.book[\mathcal{X}]$ is actually an abbreviation for $book[\mathcal{X}] \mid \mathbf{T}$. The $BOOKS \vDash book[\mathcal{X}] \mid \mathbf{T}$ statement means: *BOOKS* can be split in two parts, one that satisfies $book[\mathcal{X}]$, the other one that satisfies $\mathbf{T}$. Every piece of data satisfies $\mathbf{T}$ (*True*), while only an element $book[\ldots]$ satisfies $book[\mathcal{X}]$; hence, $BOOKS \vDash book[\mathcal{X}] \mid \mathbf{T}$ means: there is an element $book[\mathcal{X}]$ at the top level of *BOOKS*.

In TQL a matching expression is actually a logic expression, combining matching-like and classical logical operators. For example, the following query combines path-expression-like logical operators and classical logical operators ($\forall$, $\Rightarrow$) to get schema information out of the data source. It retrieves the tags appearing into each book.

$$\textit{from} \quad \textit{BOOKS} \vDash \forall \mathcal{X}(.\textit{book}[\mathcal{X}] \Rightarrow .\textit{book}[\mathcal{X} \wedge .x[\mathbf{T}]]) \quad \textit{select} \quad \textit{tag}[x]$$

The query can be read as: get $\textit{tag}[x]$ for those labels x such that, for each book $\textit{book}[\mathcal{X}]$, x is the tag of one of the elements of the book. Observe how the free variable x carries information from the binder to the result. The same property is expressed below using negation, as 'there exists no *book* where x is not a sub-tag. For more examples, see [6, 10].

$$\textit{from} \quad \textit{BOOKS} \vDash \neg\ .\textit{book}[\neg\ .x[\mathbf{T}]] \quad \textit{select} \quad \textit{tag}[x]$$

3. TQL data model

Every query, and every piece of data, in TQL denotes an *information tree*. An information tree (over a label set Λ) is an unordered tree whose edges are labelled over Λ (i.e. $a[\,b[\,]|c[\,]\,] \mid a[\,]$ would be a tree with two edges labelled by a carrying $b[\,]|c[\,]$ and the empty tree as children). We allow infinitely branching trees in the formalization, but we do not support them in the implementation. Formally, information trees are nested multisets of label-tree pairs:

Definition *For a given set of* labels Λ*, the set* $\mathcal{IT}$ *of information trees over* Λ*, ranged over by* I*, is the smallest collection such that: (a) the empty multiset,* $\{\}$*, is in* $\mathcal{IT}$*; we will use* $\mathbf{0}$ *as a notation for* $\{\}$*; (b) if* m *is in* Λ *and* I *is in* $\mathcal{IT}$ *then the singleton multiset* $\{\langle m, I\rangle\}$ *is in* $\mathcal{IT}$*; we will use* $m[I]$ *as a notation for* $\{\langle m, I\rangle\}$*; (c)* $\mathcal{IT}$ *is closed under multiset union* $\biguplus_{j\in J} M(j)$ *where* $M \in J \to \mathcal{IT}$*; we will use* $\mathrm{Par}_{j\in J}\ M(j)$ *as a notation for* $\biguplus_{j\in J} M(j)$*, and* $I \mid I'$ *for* $I \uplus I'$*.*

In examples and discussions, we will often abbreviate $m[\mathbf{0}]$ as $m[]$, or as m. We assume that Λ includes the disjoint union of each basic data type of interest.

4. TQL Syntax and Semantics

We give here only a synthetic definition of the language; for a complete formal exposition see [6], for an informal one see [10].

In the syntax below, A and B denote formulas of the tree logic, Q denotes queries, and the symbol $\sim$ denotes a binary operator belonging to a fixed set of label comparison operators, such as $=$, $\leq$, closed under negation.

In a query $f(Q)$, the function f is chosen from a fixed set $\boldsymbol{f}_{set}$ of functions of type $\mathcal{IT} \to \mathcal{IT}$. $\boldsymbol{f}_{set}$ includes functions such as *count*, that returns the information tree $n[\mathbf{0}]$ when is applied to an information tree with n elements. To simplify notation, we skip the distinction between functions and their syntactical representation.

In a formula A, variables that are not bound by $\exists x$, $\exists \mathcal{X}$, or $\mu\xi$, are *free* in A, and they are used (as $\mathcal{X}$ and x in previous examples) to pass information from the binder ($Q \vDash A$) to the query result. Thus, in *from* $Q \vDash A$ *select* Q' all the label and tree variables that are free in A are (by definition) bound in the scope Q' (i.e., they score as bound variables when we consider the whole from-select expression). In the syntax below, we write E_v whenever the variable v is bound in the scope E, as in $\exists \mathcal{X}.A_{\mathcal{X}}$ or in *from* $Q \vDash A$ *select* $Q'_{FV(A)}$. Finally, a binder $Q \vDash A$ is only well formed when no recursion variable ξ is free in A.

Hereafter, we use $\boldsymbol{A}_{set}$ to denote the set of all formulae A, $\boldsymbol{x}_{set}$ the set of label variables x, $\boldsymbol{\mathcal{X}}_{set}$ the set of tree variables $\mathcal{X}$ and $\boldsymbol{\xi}_{set}$ the set of recursive variables ξ, and similarly for any other syntactic entity in Section 5.

TQL syntax

$$
\begin{array}{lll}
L & ::= & n \mid x \\
A, B & ::= & \mathbf{0} \mid L[A] \mid A|B \mid \mathbf{T} \mid \neg A \mid A \wedge B \mid \mathcal{X} \mid \exists x.A_x \mid \exists \mathcal{X}.A_{\mathcal{X}} \mid L \sim L' \mid \xi \mid \mu\xi.A_\xi \\
Q & ::= & \textit{from } Q \vDash A \textit{ select } Q'_{FV(A)} \mid \mathcal{X} \mid \mathbf{0} \mid Q|Q' \mid L[Q] \mid f(Q)
\end{array}
$$

A formula $\mu\xi.A_\xi$ is well formed when ξ only appears positively in A_ξ.

The interpretation of a formula A, i.e. the set of all information trees that satisfy A, is only defined with respect to a pair of valuations ρ and δ that give a value to the free variables of A. The valuation ρ maps label variables x to labels (elements of Λ) and tree variables $\mathcal{X}$ to information trees, while δ maps recursion variables ξ to sets of information trees. This interpretation is defined by the map $[\![A]\!]_{\rho,\delta}$, as specified in the table below.

To simplify the notation in the comparison case, we define the ρ's extension ρ^+ by fixing $\rho^+(n) = n$ for each $n \in \Lambda$ and $\rho^+(x) = \rho(x)$; hence, we can express in one line all the four cases of label comparison.

Tree Logic: formulas as sets of information trees

$$
\begin{array}{lll|lll}
[\![\mathbf{0}]\!]_{\rho,\delta} & =_{def} & \{\mathbf{0}\} & [\![L[A]]\!]_{\rho,\delta} & =_{def} & \{\rho^+(L)[I] \mid I \in [\![A]\!]_{\rho,\delta}\} \\
[\![\mathbf{T}]\!]_{\rho,\delta} & =_{def} & \mathcal{IT} & [\![A \mid B]\!]_{\rho,\delta} & =_{def} & \{I \mid I' \mid I \in [\![A]\!]_{\rho,\delta}, I' \in [\![B]\!]_{\rho,\delta}\} \\
[\![\neg A]\!]_{\rho,\delta} & =_{def} & \mathcal{IT} \setminus [\![A]\!]_{\rho,\delta} & [\![A \wedge B]\!]_{\rho,\delta} & =_{def} & [\![A]\!]_{\rho,\delta} \cap [\![B]\!]_{\rho,\delta} \\
[\![\exists x.A]\!]_{\rho,\delta} & =_{def} & \bigcup_{n \in \Lambda} [\![A]\!]_{\rho[x \mapsto n],\delta} & [\![\exists \mathcal{X}.A]\!]_{\rho,\delta} & =_{def} & \bigcup_{I \in \mathcal{IT}} [\![A]\!]_{\rho[\mathcal{X} \mapsto I],\delta} \\
[\![\mathcal{X}]\!]_{\rho,\delta} & =_{def} & \{\rho(\mathcal{X})\} & [\![L \sim L']\!]_{\rho,\delta} & =_{def} & \text{if } \rho^+(L) \sim \rho^+(L') \text{ then } \mathcal{IT} \text{ else } \emptyset \\
[\![\xi]\!]_{\rho,\delta} & =_{def} & \delta(\xi) & [\![\mu\xi.A]\!]_{\rho,\delta} & =_{def} & \bigcap \{S \subseteq \mathcal{IT} \mid S \supseteq [\![A]\!]_{\rho,\delta[\xi \mapsto S]}\}
\end{array}
$$

We say that an information tree I satisfies a formula A with respect to ρ, δ, and write $I \vDash_{\rho,\delta} A$, when $I \in [\![A]\!]_{\rho,\delta}$. The definition above can be read, in terms of satisfaction with respect to ρ, δ, as follows. $\mathbf{0}$ is only satisfied by the information tree $\mathbf{0}$. $L[A]$ is satisfied by $m[I]$, if $m = \rho^+(L)$ and I satisfies A.$\mathbf{T}$ is satisfied by any I. $A' \mid A''$ is satisfied by I iff there exist I' and I'' such that $I' \mid I'' = I$ (where $\mid$ is multiset union) and I' satisfies A', and I'' satisfies A''. $\neg A$ is classical negation: it is satisfied by I iff I does not satisfy A. $A' \wedge A''$ is satisfied by I iff I satisfies both A' and A''. I satisfies $\exists x.A$ iff there exists some value n for x such that I is in $[\![A]\!]_{\rho[x \mapsto n],\delta}$. Here $\rho[x \mapsto n]$ denotes the valuation that maps x to n and otherwise coincides with ρ. $[\![L \sim L']\!]_{\rho,\,\delta}$ is the set $\mathcal{IT}$ if the comparison holds (w.r.t. ρ), else it is the empty set. $\mu\xi.A$ is satisfied by I iff I satisfies $A\{\xi \leftarrow \mu\xi.A\}$. Formally, $[\![\mu\xi.A]\!]_{\rho,\,\delta}$ is the least fix-point (with respect to set inclusion) of the function that maps any set of information trees S to $[\![A]\!]_{\rho,\,\delta[\xi\mapsto S]}$; the function is monotonic since any path from ξ to its binder is required to contain an even number of negations.

Valuations are the "pattern matching" mechanism of our query language; for example, $m[n[\mathbf{0}]]$ is in $[\![x[\mathcal{X}]]\!]_{\rho,\,\delta}$ if ρ maps x to m and $\mathcal{X}$ to $n[\mathbf{0}]$. We call *binding process* the process of finding all possible ρ's such that $I \in [\![A]\!]_{\rho,\,\delta}$. The implementation of the binding process is the core of the TQL processor.

The semantics of a query is defined in the following table. A query is evaluated with respect to an *input valuation* ρ, initialized with bindings for all the reachability roots of the database and also used to pass information from the surrounding *from-select* clauses.

from-select is the interesting case. Here, the sub-query Q' is evaluated once for each valuation ρ' obtained by the binding process for the formula A and the tree Q, that is once for each valuation ρ' that extends the input valuation ρ ($\rho' \supseteq \rho$) and such that $[\![Q]\!]_\rho \in [\![A]\!]_{\rho',\,\epsilon}$ (δ is initialized with the empty valuation ϵ since no recursion variable can be free in A); all the resulting trees are then combined using the $|$ operator.

Query semantics

$$[\![\mathbf{0}]\!]_\rho =_{def} \mathbf{0} \qquad [\![\mathcal{X}]\!]_\rho =_{def} \rho(\mathcal{X}) \qquad [\![m[Q]]\!]_\rho =_{def} m[[\![Q]\!]_\rho]$$

$$[\![x[Q]]\!]_\rho =_{def} \rho(x)[[\![Q]\!]_\rho] \qquad [\![f(Q)]\!]_\rho =_{def} f([\![Q]\!]_\rho) \qquad [\![Q \mid Q']\!]_\rho =_{def} [\![Q]\!]_\rho \mid [\![Q']\!]_\rho$$

$$[\![\mathit{from}\ Q \vDash A \mathit{select}\ Q']\!]_\rho =_{def} \mathit{Par}_{\rho' \in \{\rho' \mid\ dom(\rho')=dom(\rho)\cup FV(A),\ \rho'\supseteq\rho,\ [\![Q]\!]_\rho \in [\![A]\!]_{\rho',\,\epsilon}\}} [\![Q']\!]_{\rho'}$$

As usual, negation allows us to derive useful 'dual' logical operators, such as universal quantification and disjunction. In [9] we describe the semantics and implementation of such operators and of *path formulas*, the derived logical operators that allow the programmer to retrieve information found at the end, or in the middle, of any path described by a regular expression over labels.

5. TQL Algebra

As happens with any declarative query language, TQL queries are translated into an algebraic form before execution. They are translated into terms of *TQL Algebra*, an algebra with two main sorts, tables and information trees, that are used to translate, respectively, binders and queries. Binder translation performs a "semantic inversion": it transforms the operators of the tree logic, whose terms denote functions from a valuation to a set of trees, into algebraic operators, that receive a tree and return a set of valuations (a *table*). For example, a formula $x[\mathbf{T}]$, that denotes the function $\lambda\rho.\ \{\rho(x)[I] \mid I \in \mathcal{IT}\}$, is translated into the table expression *if* $\mathcal{Q} = y[\mathcal{Y}]$ *then* $\{(x \mapsto y)\}$ *else* $\mathbf{0}$ that (informally) for each tree denoted by $\mathcal{Q}$, returns the set containing the valuation $(x \mapsto m)$ if $\mathcal{Q} = m[I]$ for some m, I, and the empty table otherwise.

TQL Algebra has been defined as a tool to translate TQL but is quite natural and general. The table operators are essentially the standard relational operators [1], generalized to infinite tables and to admit $\mathcal{IT}$ as a domain. The only new operators are the two operators, *if* and $\bigcup$, that are needed to build a table depending on the structure of the input information tree, and two more that are used to define and apply recursive functions. The tree expressions exactly mirror operators used to build an information tree, plus the operator that, mirroring the behavior of from-select, uses a table to build a tree.

In this section we will present the syntax and semantics of TQL Algebra; in the next sections we will show how TQL is translated into the algebra, and how the algebra is implemented.

5.1. Algebra Sorts and their Semantics

The example above shows how the TQL variables x and $\mathcal{X}$ become, in the algebra, the field names of the *rows* of the algebraic tables (i.e., the column names), while new algebraic variables (y, $\mathcal{Y}$) are introduced. Hence, in this section, the term *variable* will refer to the algebraic variables y, $\mathcal{Y}$, while x and $\mathcal{X}$ will be called *row field names*. Hereafter, the metavariable V will stand for either a field name $\mathcal{X}$, whose *universe* $U(\mathcal{X})$ is defined to be the set $\mathcal{IT}$ of all information trees, or a field name x, whose *universe* $U(x)$ is defined to be the set Λ of all labels. The metavariable $\mathbf{V}$ will range over schemas, i.e. finite sequences $V_1, \ldots, V_n$.

The query algebra is based on four sorts: a sort of *row expressions*, ranged over by $\mathcal{R}$ or $\mathcal{R}^{\mathbf{V}}$, a sort of *label expressions*, ranged over by $\mathcal{L}$, a sort of *table expressions*, ranged over by $\mathcal{T}$ and $\mathcal{T}^{\mathbf{V}}$, and a sort of *tree expressions*, ranged over by $\mathcal{Q}$.

A row expression $\mathcal{R}^{\mathbf{V}}$ denotes a *row* (or *valuation*) over $\mathbf{V}$, that is a function that maps each $V \in \mathbf{V}$ to an element of $U(V)$ (such as $(x \mapsto m,\ \mathcal{X} \mapsto m[\mathbf{0}])$, if $\mathbf{V} = \{x, \mathcal{X}\}$). $1^{\mathbf{V}}$ will denote the set of all rows having schema $\mathbf{V}$.

A table expressions $\mathcal{T}^{\mathbf{V}}$ denotes a finite or infinite table with schema $\mathbf{V}$, that is a set of rows over $\mathbf{V}$. A table expression is used to represent the evaluation of a TQL binding operation $Q \vDash A$; this evaluation returns a set of valuations with the same schema. Hence, $\mathcal{P}(1^{\mathbf{V}})$ denotes the set of all tables with schema $\mathbf{V}$. The set $\mathcal{P}(1^{\mathbf{V}})$ contains two special tables: $0^{\mathbf{V}}$, the *empty* table with schema $\mathbf{V}$, and $1^{\mathbf{V}}$, the *full* table with schema $\mathbf{V}$. Finally, a label expression $\mathcal{L}$ denotes an element of Λ, and a tree expression $\mathcal{Q}$ denotes an element of $\mathcal{IT}$.

5.2. Syntax

The syntax is presented in the table below. The algebra variables are r, y, $\mathcal{Y}$, M. Pedices are used to specify where variables are bound: for example, in *letrec* $M = \lambda\mathcal{Y}.\ \mathcal{T}_{M,\mathcal{Y}}$ *in* $\mathcal{T}'_M$, M is bound in both $\mathcal{T}_{M,\mathcal{Y}}$ and $\mathcal{T}'_M$, while $\mathcal{Y}$ is bound in $\mathcal{T}_{M,\mathcal{Y}}$ only.

As shown in the table below, the TQL Algebra has two forms of row expressions: the variable row expression $r^{\mathbf{V}}$, and the concatenation of two row expressions. Row variables arise during the translation of *from* $Q \vDash A$ *select* Q' queries, and range over the valuations obtained by the binding $Q \vDash A$; in the algebra, a row variable is bound by the operator $Par_{r \in \mathcal{T}}\ \mathcal{Q}_r$.

The TQL Algebra has three label expressions. $\mathcal{R}(x)$ extracts a label field x from $\mathcal{R}$; m is a label constant; y is a label variable, bound by the *if* operator.

The TQL Algebra has three operators to build one-row tables, that are $\{\mathcal{R}^{\mathbf{V}}\}$, $\{(x \mapsto \mathcal{L})\}$, and $\{(\mathcal{X} \mapsto \mathcal{Q})\}$: $\{\mathcal{R}^{\mathbf{V}}\}$ denotes a table, with schema $\mathbf{V}$, only containing the row denoted by $\mathcal{R}^{\mathbf{V}}$. $\{(x \mapsto \mathcal{L})\}$ and $\{(\mathcal{X} \mapsto \mathcal{Q})\}$ both denote a table with one row and one column only, mapping, respectively, x to the label denoted by $\mathcal{L}$, $\mathcal{X}$ to the denotation of $\mathcal{Q}$.

Then we have six table operators: universe (denoting the full table $1^{\mathbf{V}}$), union, cartesian product, projection, complement, and restriction, each carrying schema information. They correspond to standard operations of relational algebra. Restriction $\sigma^{\mathbf{V}}_{\mathcal{C} \sim \mathcal{C}'} \mathcal{T}^{\mathbf{V}}$ is subtle, since each argument $\mathcal{C}$ and $\mathcal{C}'$ of the comparison may be either a label $\mathcal{L}$ or a field name x. When at least one argument is a field name x, then x must appear in the schema $\mathbf{V}$ of $\mathcal{T}^{\mathbf{V}}$, and restriction is used to select a subset of the rows of $\mathcal{T}^{\mathbf{V}}$, depending on the value of their x field. In the special case when both arguments are label expressions, restriction returns either the whole $\mathcal{T}^{\mathbf{V}}$, if the comparison succeeds, or an empty table, if the comparison fails (evaluates to false).

Then, the table algebra contains two operators that analyze a tree and build a table according to its structure; the first (*if*) analyzes the vertical structure $m[I]$ of a singleton information tree, and the second analyzes the horizontal structure of an information tree, by evaluating an expression $\mathcal{T}_{\mathcal{Y}',\mathcal{Y}''}$ for each horizontal partition $\mathcal{Y}' \mid \mathcal{Y}''$ of the information tree denoted by $\mathcal{Q}$.

Finally, the table algebra has two operators used to translate recursive formulas: *letrec* $M = \lambda\mathcal{Y}.\ \mathcal{T}_{M,\mathcal{Y}}$ *in* $\mathcal{T}'_M$ computes the least fix-point of the monotone function $\lambda M.(\lambda\mathcal{Y}.\ \mathcal{T}_{M,\mathcal{Y}})$, in the space of functions from trees to tables, while $M(\mathcal{Q})$ applies such a fix-point to a tree.

The tree algebra reflects the TQL operators used to build trees. The essential difference is that here $\mathcal{X}$ does not denote a variable but the name of a field in the row ρ, while we have a new metavariable $\mathcal{Y}$, ranging over the tree variables. A variable $\mathcal{Y}$ is bound by the *if*, $\bigcup$, and *letrec* operators.

Query algebra, primitive operators:

$\mathcal{R}^{\mathbf{V}} ::=$	row expression
$\quad r^{\mathbf{V}}$	variable row expression ($r^{\mathbf{V}} \in \boldsymbol{r}_{set}$)
$\quad \mathcal{R}'^{\mathbf{V}'}; \mathcal{R}''^{\mathbf{V}''}$	row concatenation ($\mathbf{V}' \cap \mathbf{V}'' = \emptyset,\ \mathbf{V}' \cup \mathbf{V}'' = \mathbf{V}$)
$\mathcal{L} ::=$	label expression
$\quad y$	label variable ($y \in \boldsymbol{y}_{set}$)
$\quad m$	label
$\quad \mathcal{R}^{\mathbf{V}}(x)$	field extraction from the row ($x \in \mathbf{V}$)
$\mathcal{C} ::=$	comparison argument in the restriction operator
$\quad x$	row field name ($x \in \boldsymbol{x}_{set}$)
$\quad \mathcal{L}$	label expression

$\mathcal{T}^{\mathbf{V}} ::=$	table expression
$\{\mathcal{R}^{\mathbf{V}}\}$	one-row table
$\{(x \mapsto \mathcal{L})\}$	singleton: one column/one row ($\mathbf{V} = \{x\}$)
$\{(\mathcal{X} \mapsto \mathcal{Q})\}$	singleton ($\mathbf{V} = \{\mathcal{X}\}$)
$\mathbf{1}^{\mathbf{V}}$	universe: every row over $\mathbf{V}$
$\mathcal{T}^{\mathbf{V}} \cup^{\mathbf{V}} \mathcal{T}'^{\mathbf{V}}$	binary union
$\mathcal{T}'^{\mathbf{V}'} \times^{\mathbf{V}',\mathbf{V}''} \mathcal{T}''^{\mathbf{V}''}$	cartesian product ($\mathbf{V}' \cap \mathbf{V}'' = \emptyset,\ \mathbf{V}' \cup \mathbf{V}'' = \mathbf{V}$)
$\prod_{\mathbf{V}}^{\mathbf{V}'} \mathcal{T}^{\mathbf{V}'}$	projection ($\mathbf{V} \subseteq \mathbf{V}'$)
$Co^{\mathbf{V}}(\mathcal{T}^{\mathbf{V}})$	complement
$\sigma^{\mathbf{V}}_{\mathcal{C} \sim \mathcal{C}'} \mathcal{T}^{\mathbf{V}}$	restriction
$\mathit{if}\ \mathcal{Q} = y[\mathcal{Y}]\ \mathit{then}\ \mathcal{T}^{\mathbf{V}}_{\mathcal{Y},y}\ \mathit{else}\ \mathcal{T}'^{\mathbf{V}}$	test for $y[\mathcal{Y}]$
$\bigcup^{\mathbf{V}}_{\{\mathcal{Y}' \mid \mathcal{Y}'' = \mathcal{Q}\}} \mathcal{T}_{\mathcal{Y}',\mathcal{Y}''}$	union of $\mathcal{T}_{\mathcal{Y}',\mathcal{Y}''}$ for decompositions $\mathcal{Y}' \mid \mathcal{Y}''$ of $\mathcal{Q}$
$\mathit{letrec}\ M^{\mathbf{V}'} = \lambda\mathcal{Y}.\ \mathcal{T}'^{\mathbf{V}'}_{M^{\mathbf{V}'},\mathcal{Y}}$ $\mathit{in}\ \mathcal{T}^{\mathbf{V}}_{M^{\mathbf{V}'}}$	recursive definition of a function ($M^{\mathbf{V}'} \in \boldsymbol{M}_{set}$) from trees ($\mathcal{Y}$) to tables; $M^{\mathbf{V}'}$ appears positively in $\mathcal{T}'^{\mathbf{V}'}$
$M^{\mathbf{V}}(\mathcal{Q})$	application of a recursive function to a tree
$\mathcal{Q} ::=$	tree expression
$Par_{r^{\mathbf{V}} \in \mathcal{T}^{\mathbf{V}}}\ \mathcal{Q}_{r^{\mathbf{V}}}$	union of $\mathcal{Q}_{r^{\mathbf{V}}}$ computed once for each $r^{\mathbf{V}}$ of $\mathcal{T}^{\mathbf{V}}$
$\mathcal{Y}$	tree variable ($\mathcal{Y} \in \boldsymbol{\mathcal{Y}}_{set}$)
$\mathcal{R}^{\mathbf{V}}(\mathcal{X})$	field extraction from the row $\mathcal{R}$ ($\mathcal{X} \in \mathbf{V}$)
$\mathbf{0}$	empty tree
$\mathcal{Q} \mid \mathcal{Q}'$	binary union
$\mathcal{L}[\mathcal{Q}]$	singleton tree
$f(\mathcal{Q})$	predefined function application

5.3. Semantics of TQL Algebra

Algebra expressions are evaluated with respect to an environment e, that associates each free label variable with a label, each free tree variable with an information tree, each free recursive variable with a function from trees to tables, and each row variable with a row of the right type. Formally, the type $\boldsymbol{e}_{sem}$ of e is defined as follows, where $(T \xrightarrow{p} U) \times (T' \xrightarrow{p} U')$ is the type of partial functions mapping T to U and T' to U', and $dom(f) = K$ means that $f : K \to K'$.

$$
\begin{array}{lcl}
\boldsymbol{\mathcal{R}}_{sem} & = & (\boldsymbol{x}_{set} \xrightarrow{p} \Lambda) \times (\boldsymbol{\mathcal{X}}_{set} \xrightarrow{p} \mathcal{IT}) \\
\boldsymbol{\mathcal{R}}^{\mathbf{V}}{}_{sem} & = & \{\rho \mid \rho \in \boldsymbol{\mathcal{R}}_{sem},\ dom(\rho) = \mathbf{V}\} \\
\boldsymbol{\mathcal{T}}_{sem} & = & \cup_{\mathbf{V}} \mathcal{P}(1^{\mathbf{V}}) \\
\boldsymbol{e}'_{sem} & = & (\boldsymbol{y}_{set} \xrightarrow{p} \Lambda) \times (\boldsymbol{\mathcal{Y}}_{set} \xrightarrow{p} \mathcal{IT}) \times (\boldsymbol{M}_{set} \xrightarrow{p} \mathcal{IT} \xrightarrow{p} \boldsymbol{\mathcal{T}}_{sem}) \times (\boldsymbol{r}_{set} \xrightarrow{p} \boldsymbol{\mathcal{R}}_{sem}) \\
\boldsymbol{e}_{sem} & = & \{e \mid e \in \boldsymbol{e}'_{sem},\ \forall r^{\mathbf{V}} \in dom(e).\ e(r^{\mathbf{V}}) \in \boldsymbol{\mathcal{R}}^{\mathbf{V}}{}_{sem}\}
\end{array}
$$

The type of the semantic function $\lambda e.\lambda\_.\, [\![\_]\!]_e$ is: $\boldsymbol{e}_{sem} \to ((\boldsymbol{\mathcal{R}}_{set} \xrightarrow{p} \boldsymbol{\mathcal{R}}_{sem}) \times (\boldsymbol{\mathcal{L}}_{set} \xrightarrow{p} \Lambda) \times (\boldsymbol{\mathcal{Q}}_{set} \xrightarrow{p} \mathcal{IT}) \times (\boldsymbol{\mathcal{T}}_{set} \xrightarrow{p} \boldsymbol{\mathcal{T}}_{sem}))$. Most of the algebra semantics is straightforward. The crucial point is the information flow among the different sorts: tables depend on tree expressions when a singleton table is built, and when a tree analysis operation (*if* or $\bigcup$) is performed. Information flows from table expressions to trees when $Par_{r\in\mathcal{T}}\ \mathcal{Q}_r$ is evaluated, and it flows through the row variable r.

letrec involves the computation of a minimal fixpoint, that is well-defined since the function mapping f' to $\lambda t : \mathcal{IT}.\llbracket\mathcal{T}\rrbracket_{e[\mathcal{Y}\mapsto t][M^{\mathbf{V}}\mapsto f']}$ is monotone, because $M^{\mathbf{V}}$ only appears positively inside $\mathcal{T}$. We report here a couple of cases, while the full table is in [9].

The semantics of some table and tree expressions:

$$
\begin{array}{lcl}
\llbracket\sigma^{\mathbf{V}}_{x\sim\mathcal{L}}\mathcal{T}^{\mathbf{V}}\rrbracket_e & = & \{\rho \mid \rho\in\llbracket\mathcal{T}^{\mathbf{V}}\rrbracket_e,\ \rho(x)\sim\llbracket\mathcal{L}\rrbracket_e\} \\
\llbracket \mathit{letrec}\ M^{\mathbf{V}} = \lambda\mathcal{Y}.\ \mathcal{T}\ \mathit{in}\ \mathcal{T}'\rrbracket_e & = & \mathit{let}\ f = \mathit{minfix}_{f'}(\lambda t : \mathcal{IT}.\llbracket\mathcal{T}\rrbracket_{e[\mathcal{Y}\mapsto t][M^{\mathbf{V}}\mapsto f']}) \\
 & & \quad \mathit{in}\ \ \llbracket\mathcal{T}'\rrbracket_{e[M^{\mathbf{V}}\mapsto f]} \\
\llbracket M^{\mathbf{V}}(\mathcal{Q})\rrbracket_e & = & e(M^{\mathbf{V}})(\llbracket\mathcal{Q}\rrbracket_e) \\
\llbracket Par_{r\in\mathcal{T}}\ \mathcal{Q}\rrbracket_e & = & Par_{\rho\in\llbracket\mathcal{T}\rrbracket_e}\ \llbracket\mathcal{Q}\rrbracket_{e[r\mapsto\rho]}
\end{array}
$$

5.4. Derived Operators

As for TQL, in the actual system there are many more algebraic operators that can be defined in terms of the primitive ones. In the translation of base operators of TQL (Section 6) the only derived operators we use are some variants of *if* and a generalized *natural join* operator $\mathcal{T}^{\mathbf{V}} \bowtie \mathcal{T}'^{\mathbf{V}'}$ (defined in [9]).

6. Translation of TQL into TQL Algebra

The translation of a formula A is the kernel of the translation problem. By definition, a formula defines a function from a substitution to a set of trees, but we want to transform it into an algebraic expression which, applied to a tree, yields a set of substitutions.

In detail, we assume that the algebraic expressions $\mathcal{Q}$ and $\mathcal{R}$, that compute the database Q and the input substitution ρ, are given. We define now the translation $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\epsilon}$ of a formula A as an algebraic expression that computes the set of all substitutions ρ' such that $\llbracket Q\rrbracket_\rho \vDash_{(\rho;\rho'),\,\epsilon} A$, i.e. the rows that are passed from the binder $Q \vDash A$ to the *select* branch of a query. In order to deal with recursive formulae, we have to add a third parameter $\gamma : \boldsymbol{\xi}_{set} \xrightarrow{\mathrm{P}} \boldsymbol{M}_{set}$ that maps logical recursive variable to the algebraic ones. For each γ, $\hat{\gamma}$ describes its schema, i.e. $\hat{\gamma}(\xi) = \mathbf{V} \Leftrightarrow \gamma(\xi) = M^{\mathbf{V}}$, for some M.

The translation $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ depends on a function $\mathcal{S}(A,\mathbf{V},\hat{\gamma})$ that computes the schema of $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ (essentially, it subtracts $\mathbf{V}$ from $FV(A)$, but recursion is subtle; see [9]).

We shortly describe some cases of the binder translation algorithm. $\mathbf{T}$ returns the table $\mathbf{1}^{\emptyset}$, the table that, when joined to any other table, does not exclude any row. A clause $Q \vDash \mathbf{0}$ returns the table $\mathbf{1}^{\emptyset}$ if Q is $\mathbf{0}$, and the table $\mathbf{0}^{\emptyset}$ otherwise. The operator $\wedge$ is interpreted by table join. The translation $\lceil\!\lceil \mathcal{X}\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ depends on which variables are already bound by $\mathcal{R}^{\mathbf{V}}$: if $\mathcal{X}$ belongs to $\mathbf{V}$, then the resulting expression must have an empty schema, hence can only be $\mathbf{1}^{\emptyset}$ or $\mathbf{0}^{\emptyset}$ (depending on whether $\mathcal{Q}$ coincides with $\mathcal{R}^{\mathbf{V}}(\mathcal{X})$); otherwise, the expression must denote a table mapping $\mathcal{X}$ to $\mathcal{Q}$. $\lceil\!\lceil n[A]\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ denotes a table $\mathbf{0}^{\mathcal{S}(A,\mathbf{V},\hat{\gamma})}$ if the information tree denoted by $\mathcal{Q}$ does not match $n[I']$, otherwise it denotes the same table as $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma}$, evaluated in an environment that binds $\mathcal{Y}$ to I'. If x is in $\mathbf{V}$, the translation $\lceil\!\lceil x[A]\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ goes essentially as in the previous case. Otherwise, the table denoted by $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma}$ has to be joined with the one mapping x to the label y (table join corresponds to conjunction). $\lceil\!\lceil \neg A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ is the table with all the rows that do not satisfy $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$. Comparison translation uses the abbreviation $\mathcal{R}^{\mathbf{V}}_{+}(L)$, which stands for $\mathcal{R}^{\mathbf{V}}(L)$ if $L \in \mathbf{V}$, and for L otherwise (when L is a variable not in $\mathbf{V}$, or a constant). $\lceil\!\lceil A \mid B\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ is executed by considering every possible decomposition $I' \mid I''$ of the denotation of $\mathcal{Q}$, and by collecting all rows that satisfy both $I' \vDash A$ and $I'' \vDash B$. The nesting $\bigcup(\_ \bowtie \_)$ of the algorithm corresponds to the nesting "**there exists** a decomposition $I' \mid I''$ such that **both** $I' \vDash A$ **and** $I'' \vDash B$ hold". For existential quantification, $\lceil\!\lceil \exists\mathcal{X}.\ A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ can be computed as $\prod_{FV(A)\setminus\{\mathcal{X}\}} \lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$, because an information tree I belongs to $[\![\exists\mathcal{X}.A]\!]_{\rho,\,\delta}$ if, for some I', $I \in [\![A]\!]_{\rho[\mathcal{X}\mapsto I'],\,\delta}$.

Observe that the translation actually depends only on the shape of A and on the schema $\mathbf{V}$ of $\mathcal{R}^{\mathbf{V}}$, while $\mathcal{Q}$, $\mathcal{R}^{\mathbf{V}}$ and γ are only 'plugged' somewhere, without ever analyzing their shape.

The translation of recursion is the trickiest bit. In $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma[\xi\mapsto M]}$, the M variable corresponds to ξ, hence it means 'here you evaluate the translation of A again'. However, in general, you have to evaluate it against a different tree, since some of the logical operations (and their algebraic counterparts) 'walk' inside the input database; for example, $m[I] \vDash m[A]$ is reduced to $I \vDash A$, changing both the formula and the model ($m[A] \to A$, $m[I] \to I$). For this reason, the translation process $\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}$ analyzes A and produces a translation by keeping track, at any time, of the 'current tree expression' $\mathcal{Q}$. Therefore, the translation of the recursion body A is performed parametrically with respect to the actual input tree ($\lambda\mathcal{Y}.\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma[\xi\mapsto M]}$), and, whenever ξ is met, the corresponding M (i.e. $\gamma(\xi)$) is applied to the current input tree $\mathcal{Q}$.

Binder and query translation

$$
\begin{array}{lll l}
\lceil\!\lceil \mathbf{T}\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathbf{1}^{\emptyset} & \\
\lceil\!\lceil \mathbf{0}\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathit{if}\ \mathcal{Q} = \mathbf{0}\ \mathit{then}\ \mathbf{1}^{\emptyset}\ \mathit{else}\ \mathbf{0}^{\emptyset} & \\
\lceil\!\lceil A \wedge B\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} \bowtie^{\mathcal{S}(A,\mathbf{V},\hat{\gamma}),\,\mathcal{S}(B,\mathbf{V},\hat{\gamma})} \lceil\!\lceil B\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & \\
\lceil\!\lceil \mathcal{X}\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathit{if}\ \mathcal{Q} = \mathcal{R}(\mathcal{X})\ \mathit{then}\ \mathbf{1}^{\emptyset}\ \mathit{else}\ \mathbf{0}^{\emptyset} & \text{if } \mathcal{X} \in \mathbf{V} \\
\lceil\!\lceil \mathcal{X}\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \{(\mathcal{X} \mapsto \mathcal{Q})\} & \text{if } \mathcal{X} \notin \mathbf{V} \\
\lceil\!\lceil n[A]\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathit{if}\ \mathcal{Q} = n[\mathcal{Y}]\ \mathit{then}\ \lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma}\ \mathit{else}\ \mathbf{0}^{\mathcal{S}(A,\mathbf{V},\hat{\gamma})} & \\
\lceil\!\lceil x[A]\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathit{if}\ \mathcal{Q} = \mathcal{R}(x)[\mathcal{Y}]\ \mathit{then}\ \lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma}\ \mathit{else}\ \mathbf{0}^{\mathcal{S}(A,\mathbf{V},\hat{\gamma})} & \text{if } x \in \mathbf{V} \\
\lceil\!\lceil x[A]\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathit{if}\ \mathcal{Q} = y[\mathcal{Y}] & \text{if } x \notin \mathbf{V} \\
 & & \quad \mathit{then}\ \{(x \mapsto y)\} \bowtie^{\{x\},\,\mathcal{S}(A,\mathbf{V},\hat{\gamma})} \lceil\!\lceil A\rceil\!\rceil_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma} & \\
 & & \quad \mathit{else}\ \mathbf{0}^{\mathcal{S}(x[A],\mathbf{V},\hat{\gamma})} & \\
\lceil\!\lceil \neg A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & Co^{\mathcal{S}(A,\mathbf{V},\hat{\gamma})}(\lceil\!\lceil A\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma}) & \\
\lceil\!\lceil L \sim L'\rceil\!\rceil_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \sigma^{\mathcal{S}(L\sim L',\mathbf{V},\hat{\gamma})}_{\mathcal{R}^{\mathbf{V}}_{+}(L)\sim\mathcal{R}^{\mathbf{V}}_{+}(L')}\, \mathbf{1}^{\mathcal{S}(L\sim L',\mathbf{V},\hat{\gamma})} &
\end{array}
$$

$$
\begin{array}{lll}
\llbracket A \mid B \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \bigcup^{\mathcal{S}(A|B,\mathbf{V},\hat{\gamma})}_{\{\mathcal{Y}' | \mathcal{Y}''=\mathcal{Q}\}} (\llbracket A \rrbracket_{\mathcal{Y}',\mathcal{R}^{\mathbf{V}},\gamma} \ \bowtie^{\mathcal{S}(A,\mathbf{V},\hat{\gamma}),\, \mathcal{S}(B,\mathbf{V},\hat{\gamma})} \llbracket B \rrbracket_{\mathcal{Y}'',\mathcal{R}^{\mathbf{V}},\gamma}) \\
\llbracket \exists x.\ A \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \prod^{\mathcal{S}(A,\mathbf{V},\hat{\gamma})}_{\mathcal{S}(A,\mathbf{V},\hat{\gamma})\setminus\{x\}} \llbracket A \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} \\
\llbracket \exists \mathcal{X}.\ A \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \prod^{\mathcal{S}(A,\mathbf{V},\hat{\gamma})}_{\mathcal{S}(A,\mathbf{V},\hat{\gamma})\setminus\{\mathcal{X}\}} \llbracket A \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} \\
\llbracket \mu\xi.A \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \mathit{letrec}\ M^{\mathcal{S}(\mu\xi.A,\mathbf{V},\hat{\gamma})} = \lambda\mathcal{Y}.\llbracket A \rrbracket_{\mathcal{Y},\mathcal{R}^{\mathbf{V}},\gamma[\xi\mapsto M]}\, \mathit{in}\ M^{\mathcal{S}(\mu\xi.A,\mathbf{V},\hat{\gamma})}(\mathcal{Q}) \\
\llbracket \xi \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\gamma} & =_{def} & \gamma(\xi)(\mathcal{Q})
\end{array}
$$

$$
\begin{array}{llllll}
\llbracket \mathbf{0} \rrbracket_{\mathcal{R}^{\mathbf{V}}} & =_{def} & \mathbf{0} & \llbracket \mathcal{X} \rrbracket_{\mathcal{R}^{\mathbf{V}}} & =_{def} & \mathcal{R}^{\mathbf{V}}(\mathcal{X}) \\
\llbracket m[Q] \rrbracket_{\mathcal{R}^{\mathbf{V}}} & =_{def} & m[\llbracket Q \rrbracket_{\mathcal{R}^{\mathbf{V}}}] & \llbracket x[Q] \rrbracket_{\mathcal{R}^{\mathbf{V}}} & =_{def} & \mathcal{R}^{\mathbf{V}}(x)[\llbracket Q \rrbracket_{\mathcal{R}^{\mathbf{V}}}] \\
\llbracket f(Q) \rrbracket_{\mathcal{R}^{\mathbf{V}}} & =_{def} & f(\llbracket Q \rrbracket_{\mathcal{R}^{\mathbf{V}}}) & \llbracket Q \mid Q' \rrbracket_{\mathcal{R}^{\mathbf{V}}} & =_{def} & \llbracket Q \rrbracket_{\mathcal{R}^{\mathbf{V}}} \mid \llbracket Q' \rrbracket_{\mathcal{R}^{\mathbf{V}}}
\end{array}
$$

$$
\llbracket \mathit{from}\ Q \vDash A\ \mathit{select}\ Q' \rrbracket_{\mathcal{R}^{\mathbf{V}}} =_{def} \mathit{Par}_{r^{\mathbf{V}'} \in \llbracket A \rrbracket_{\mathcal{Q},\mathcal{R}^{\mathbf{V}},\epsilon}} \llbracket Q' \rrbracket_{\mathcal{R}^{\mathbf{V}};r^{\mathbf{V}'}} \quad \text{where } \mathcal{Q} = \llbracket Q \rrbracket_{\mathcal{R}^{\mathbf{V}}}
$$

The following theorem states the query translation correctness. The core of the proof is the binder translation correctness statement [9], needed in the *from-select* case.

Theorem 1 *Let* $Q \in \boldsymbol{Q}_{set}$, $e \in \boldsymbol{e}_{sem}$, *and* $\mathcal{R}^{\mathbf{V}} \in \boldsymbol{\mathcal{R}}^{\mathbf{V}}_{set}$. *Then:*

$$
FV(\mathcal{R}^{\mathbf{V}}) \subseteq dom(e),\ FV(Q) \subseteq \mathbf{V} \ \Rightarrow\ \llbracket Q \rrbracket_{e(\mathcal{R}^{\mathbf{V}})} = \lfloor\!\lfloor \llbracket Q \rrbracket_{\mathcal{R}^{\mathbf{V}}} \rfloor\!\rfloor_e
$$

7. Implementing the Algebra

The essential problem we have to face is the representation of infinite tables and trees. Our solution is not complete; we actually implement a finite representation of infinite tables, but with the following limitations: (i) we only deal with finite trees; if the user runs a query that would need to evaluate a sub-query an infinite number of times, the evaluation is aborted; (ii) we do not implement general comparison between label expressions, but only equality comparison (and its negation) when at most one of the two compared label expressions is an unbound label variable, and full comparison when none of the two label expressions is an unbound label variable; (iii) recursion evaluation may loop forever; guarded recursion (when the recursive variable is separated from its definition by a $L[\_]$ operator) is safe, however, and it seems to be expressive enough for most purposes, including all queries that use path formulas.

A finite information tree is simply represented by an implementation of nested multi-sets; we keep the semantic notation ($\mathbf{0}$, $|$, $m[I]$) for the implementation of information tree operators and of labels (m).

The implementation of tables is the interesting part, since we have to represent infinite tables, and complex operations over them. A table is represented by a structure called *disjunctive constraint*, closely related to proposals in the field of constraint databases ([14], [15]). The constraint algebra we define here, however, does not seem to have been studied before.

A *constraint* is, essentially, a table with schema $\mathbf{V}$ where each cell contains either a value or the finite representation of a cofinite set. For example, an infinite table containing $[(x \mapsto l), (\mathcal{X} \mapsto m[0])]$ and $[(x \mapsto m), (\mathcal{X} \mapsto I)]$ for any I but $l[0]$, $m[0]$ would be represented by the following constraint:

x	$\mathcal{X}$
$\{l\}$	$\{m[0]\}$
$\{m\}$	$\overline{\{l[0], m[0]\}}$

i.e.: $\{ [x{:=}\{l\}, \mathcal{X}{:=}\{m[0]\}],\ [x{:=}\{m\}, \mathcal{X}{:=}\overline{\{l[0], m[0]\}}]\}$

Formally, a *simple constraint* R is a tuple of sets S_i, each labelled with a different variable V_i, written as $[V_1 := S_1, \ldots, V_n := S_n]$, and such that: each V_i is either a label variable or a tree variable, and S_i is, respectively, a set of labels or a set of information trees; each S_i is either a singleton or the *cofinite* complement $\overline{P}$ of a finite set P. The set $dom(R) =_{def} \{V_i\}^{\,i\in I}$ is the domain of the simple constraint. Each simple constraint R defined on a domain $\mathbf{V}$ represents a *set* of rows $\rho^{\mathbf{V}}$; in detail $R = [V_1{:=}S_1, \ldots, V_n{:=}S_n]$ represents the set of all rows ρ over $\mathbf{V}$ that satisfy the constraint:

$$[\![R]\!] =_{def} \{\rho \ \mid\ \rho\in 1^{dom(R)},\ \rho(V_1)\in S_1 \wedge \ldots \wedge \rho(V_n)\in S_n\}$$

A *disjunctive constraint* $T^{\mathbf{V}}$ (or simply *constraint*) is a set of simple constraints, each with the same domain $\mathbf{V}$. It represents the union of all the sets of rows represented by its simple constraints.

Given this model, for each operator *op* defined on tables in $\boldsymbol{\mathcal{T}}_{set}$ we have to define (at least) an implementation **op** that works on disjunctive constraints. In [9] we describe them, in particular we describe original and effective algorithms for complement and coprojection (the dual operator of projection).

8. Related Works and Conclusions

There are many algebras dealing with semi-structured data and XML [11, 8, 3, 16, 13, 2], but only some of them have a documented implementation [8, 11, 13]. These algebras operate on trees and tables of trees too, although some of them represent tables as trees or forests.

However, due to the specific, logic-based, nature of TQL, none of the other algebras has the operators we need to support our language. Namely, TQL Algebra is the only one that supports: (i) *logical complement* operator, that is a complement that is not defined on the active domain of the DBMS but on the infinite sets Λ and $\mathcal{IT}$; (ii) dual operators, such as *co-projection*, that allows the translation of universal quantification even in presence of free variables; (iii) specific fix-point operators to deal with horizontal and vertical recursion; (iv) iterators allowing analysis of the horizontal structure of a forest.

Finally, due to formal approach we have taken, our algebra is the only one where the correctness of the language translation has been proved.

The work on design and implementation of TQL is far from finished. At the language level, we are currently working on (i) extensions of the language to deal with order and with trees having a superimposed graph structure; (ii) adding a type and constraint system to the language; (iii) defining a TQL sublanguage that, by exhibiting a lower expressive power, may be implemented on more standard algebras.

At the implementation level, we are working towards the design of better persistent data structures and physical operators, endowed with a cost model, to allow cost-based physical optimization. The current TQL system is available at `http://tql.di.unipi.it/tql`.

References

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

[2] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for XML. Communication to the W3C, September 1999.

[3] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB (Informal Proceedings)*, pages 37–42, 1999.

[4] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. of International World Wide Web Conference, WWW10*, May 2001.

[5] L. Cardelli. Describing semistructured data. *SIGMOD Record, Database Principles Column*, 2002. To appear.

[6] L. Cardelli and G. Ghelli. A query language based on the ambient logic. In *Proc. of European Symposium on Programming (ESOP), Genova, Italy*, April 2001.

[7] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. of POPL*. ACM Press, 2000.

[8] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD*, 1998.

[9] G. Conforti, O. Ferrara, and G. Ghelli. TQL Algebra and its Implementation. Working Draft available at http://tql.di.unipi.it/tql. Full version.

[10] G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The Query Language TQL. In *Proc. of WebDB*, 2002. To appear.

[11] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics, June 2001. W3C Working Draft.

[12] B.C. Pierce H. Hosoya. XDuce: A typed XML processing language (preliminary report). In *Proc. of Workshop on the Web and Data Bases (WebDB)*, 2000.

[13] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proceedings of DBPL'01*, 2001.

[14] P. Kanellakis. Tutorial: Constraint programming and database languages. In *Proc. of the 14th PODS*, pages 46–53. ACM Press, 1995.

[15] Peter Z. Revesz. Safe query languages for constraint databases. *ACM Transactions on Database Systems*, 23(1):58–99, March 1998.

[16] C. Sartiani and A. Albano. Yet another query algebra for XML data. In *Proc. of IEEE IDEAS*, 2002.