TQL: A Query Language for Semistructured Data Based on the Ambient Logic

Luca Cardelli
Microsoft Research

Giorgio Ghelli
Università di Pisa

Abstract

The ambient logic is a modal logic proposed to describe the structural and computational properties of distributed and mobile computation. The structural part of the ambient logic is, essentially, a logic of labeled trees, hence it turns out to be a good foundation for query languages for semistructured data, much in the same way as first order logic is a fitting foundation for relational query languages. We define here a query language for semistructured data that is based on the ambient logic, and we outline an execution model for this language. The language turns out to be quite expressive. Its strong foundations and the equivalences that hold in the ambient logic are helpful in the definition of the language semantics and execution model.

1 Introduction

This work arises from the unexpected convergence of studies in two different fields: mobile computation and semistructured data.

Unstructured collections, or unstructured data, are collections that do not respect a predefined schema, and hence need to carry a description of their own structure. These are called *semistructured* when one can recognize in them some degree of homogeneity. This partial regularity makes semistructured collections amenable to be accessed through query languages, but not through query languages that have been designed to access fully structured databases. New languages are needed that are able to tolerate the data irregularity, and that can be used to query, at the same time, both data and structure. Semistructured collections are usually modeled in terms of labeled graphs, or labeled trees [4].

The ambient logic is a modal logic proposed to describe the structural and computational properties of distributed and mobile computation [15]. The logic comes equipped with a rich collection of logical implications and equivalences. The structural part of the ambient logic is, essentially, a logic designed to describe properties of labeled trees. It is therefore a good foundation for query languages for semistructured data, much in the same way as first order logic is a fitting foundation for relational query languages.

Upon this observation we based the design of TQL, a query language for semistructured data that is based on the ambient logic.

The language turns out to be quite expressive. Moreover, the TQL subset of the ambient logic can be naturally used to express types and constraints over semistructured data. As a consequence, problems as subtyping, constraint implication, constraint satisfiability, query correctness and query containment, become special cases of the validity problem, and the same holds for their combinations, such as query containment in presence of constraints, or query correctness in presence of subtyping. 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 [26, 11]. Of course, if the full power of the logic is used, every aspect of static query analysis (correctness, containment, subtyping...) becomes undecidable, since validity of a tree-logic formula is undecidable in general. However, many decidable subsets of the logic can be defined, which are expressive enough to encode known type and constraint systems. The search for decidable subsets with the "right" balance of expressivity and cost is an open problem. This unified framework for types, constraints, and queries is a central aim of the TQL project, but its further investigation is out of the scope of this paper.

In this paper we define the query language TQL, and we outline an execution model, which is the one exploited in the current prototypical implementation of TQL.

The paper is structured as follows. In Section 2 present a preview of the query language. In Section 3 we define the tree data model. In Section 4 we present the logic, upon which the query language, presented in Section 5, is defined. In Section 6 we briefly discuss the expression of types and constraints in TQL logic. In Section 7 we present the evaluation model. In Section 8 we compare TQL with related proposals. In Section 9 we draw some conclusions.

2 TQL by examples

2.1 The Simplest Queries

We present here TQL through some examples. We begin with standard queries, borrowed from the W3C XMP Use Cases [1]. TQL queries are evaluated with respect to a global environment, defined by the user, where some variables are bound to local or remote XML files. We assume here that the variable `$Bib` has been bound to the document available at `//tql.di.unipi.it/tql/pubb.xml`, which contains bibliographic entries, as in the fragment below, written using TQL syntax:

```
bib[
  book[year[1999]
    | title[DataOnTheWeb]
    | author[ first[Serge] | last[Abiteboul] ]
    | author[ first[Dan] | last[Suciu] ]
    | author[ first[Peter] | last[Buneman] ]
    | publisher[MorganKaufmann]
    | price[45]
  ]
  book[year[1995]
    | title[FoundationsDatabases]
    | author[ first[Serge] | last[Abiteboul] ]
    | author[ first[Richard] | last[Hull] ]
    | author[ first[Victor] | last[Vianu] ]
    | publisher[Addison]
    | price[60]
  ]
  | book[year[1999]
    | title[ProceedingsICDT99]
    | editor[ first[Peter] | last[Buneman] ]
    | publisher[Springer]
    | price[12]
  ]
...
```

In this format, `bib[C]` stands for an element tagged `bib` whose contents are `C`, while `C1|C2` is the concatenation of two elements, or, more generally, of two sets of elements. TQL notation is different from XML because TQL is born as a language to query semistructured data in general, i.e. unordered trees with labelled edges; XML is just one way to construct such trees, using tagged elements (and attributes) to build labelled edges.

The basic TQL query is `from Q |= A select Q'`, where `Q` is the *subject* (or *data source*) to be matched against the formula `A`, and `Q'` is the result expression. The matching of `Q` and `A` returns a set of bindings for the variables that are free in `A`. `Q'` is evaluated once for each of these bindings, and the concatenation of the results of all these evaluations is the query result.

For example, consider the following TQL query, that returns the titles of all books written in 1999, and is evaluated in an environment where `$Bib` is bound as specified above.

```
from $Bib |= .bib[.book[.year[1999]
                        And .title[$t]
                       ]
                 ]
select title[$t]
```

The formula:

```
.bib[ .book[ .year[1999] And .title[$t] ] ]
```

is a logic formula, which should be read as: "there is a path `.bib[.book[ ]]` that reaches a place that matches `.year[1999] And .title[$t]`, i.e. a place where you find both a path `.year[]` leading to `1999` and a path `.title[ ]` leading to something, that you will call `$t`". Since `//tql.di.unipi.it/tql/pubb.xml` contains two books, with titles `DataOnTheWeb` and `ProceedingsICDT99`, the query first computes the set of bindings

{`[$t = DataOnTheWeb]; [$t = ProceedingsICDT99]`}

The subquery `title[$t]` is then evaluated once for each binding, yielding the result

`title[DataOnTheWeb]` |`title[ProceedingsICDT99]`.

The formula `.t[A]` reads "there exists an element `t` whose contents satisfy `A`"; it is actually defined in terms of three primitive operators, truth `T`, horizontal splitting `A'` | `A''`, and element matching `t[A]`.

The element formula `t[A]` only matches a one-element document: while `.t[A]` matches both trees `t[D]` and `t[D]|u[E]|...` (provided that `A` matches `D`), the formula `t[A]` only matches the first one. The truth formula `T` matches every tree. Finally, the formula $\mathtt{A}_1 | \mathtt{A}_2$ matches `D` iff `D` is equal, modulo reordering, to $\mathtt{D}_1 | \mathtt{D}_2$, with $\mathtt{A}_1$ matching $\mathtt{D}_1$ and $\mathtt{A}_2$ matching $\mathtt{D}_2$. For example, the following tree/formula pairs match, provided that `$A` is bound to `Date`:

```
title[IDB] | author[Date] | year[1994]     author[$A] | title[IDB] | year[1994]
title[IDB] | year[1994]                   T
title[IDB] | author[Date] | year[1994]     author[$A] | T
author[Date]                              author[$A] | T
```

The formula used in the last two lines can be read as "there is `author[$A]` and something else", hence it is equivalent to `.author[$A]` (the fourth pair matches since the empty tree matches `T`). For this reason, we do not take `.t[A]` as primitive, but define it as an abbreviation of `t[A]`|`T`.

The decomposition operator | is more expressive than the derived step operator `.t[A]`, since it can be also used to analyse the horizontal structure of a tree. For example, in the next query, by matching the formula `year[1999] | $EveryThingElse` against any book, we return, for any book whose year is 1999, everything but the year; `.a.b[A]` abbreviates `.a[.b[A]]`.

```
from $Bib |= .bib.book[year[1999]
                      | $EveryThingElse
                     ]
select BookOf1999[$EveryThingElse]
```

Since we have two books of 1999, there are two possible bindings for `$EveryThingElse`, each corresponding to the whole contents of a 1999 book without its `year` subedge; hence the result is:

```
BookOf1999[ title[DataOnTheWeb] | author[ first[Serge] | last[Abiteboul] ] ... ]
| BookOf1999[ title[ProceedingsICDT99] | editor[ first[Peter] | last[Buneman] ] ... ]
```

While in these examples we matched variables with trees, a TQL variable can also be matched against a tag.

For example, the following query returns any tag inside a `book` which tags an element that contains `first[Serge]` (the result is `SergeTag[author]`):

```
from $Bib |= .bib.book.$tag.first[Serge]
select SergeTag[$tag]
```

Hereafter, as a convention, we use lowercase initials for variables that are bound to tags and uppercase initials for variables that are bound to trees.

2.2 Matching and Logic

TQL logic, being a dialect of the ambient logic, contains both structural and first-order-logic operators. The structural operators (`t[A]`, $\mathtt{A}_1$ | $\mathtt{A}_2$, ...) can be used to express matching conditions, and the others can be used to combine such conditions, and to quantify over them.

For example, the condition in the following query requires the existence of a `title` field, of an `$x` field containing `Springer`, and of either an `author.last` or an `editor.last` path leading to `Buneman`.

```
from $Bib |=
     .bib.book [.title[$t]
                And Exists $x. .$x[Springer]
                And (.author.last[Buneman] Or
                     .editor.last[Buneman])
               ]
select title[$t]
```

The pattern `Exists $x. .$x[A]` is common enough to deserve the abbreviation `.%[A]`, that we will use hereafter.

Conjunction, disjunction, and existential quantification can be found in many match-based languages. TQL, however, has the full power of first-order logic, hence we can also express universal quantification and negation of arbitrary formulas. This will be discussed later.

Finally, TQL logic also includes a recursion operator that can be used, for example, to define one more derived path operator, `.%*[A]`, that matches a path of arbitrary length.

For example, the following query finds, at any nesting depth, any publication `$pub` where Suciu plays a role `$role`, and returns the title of the publication and the field where Suciu appears, preserving the tags of both.

```
from $Bib2 |= .%*.$pub[ .title[$T]
                        And .$role[.lastname[Suciu]]
                      ]
select $pub[ title[$T] | $role[Dan Suciu] ]
```

2.3 Restructuring the Data Source

In TQL, a subquery can appear wherever a tree expression is expected. This feature can be exploited to use the nesting structure of the query in order to describe the nesting structure of the result. For example, in our data source there is an entry for each book, containing the list of its authors. We can restructure it to obtain an entry for each author, containing the list of its books. The structure of the result can be visualized as follows, where `t[F]*` indicates an arbitrary repetition `t[F] | t[F] | ...` of the `t[F]` structure:

```
author[ authorname[...]
        | book[...]*
      ]*
```

Observe how this structure is reflected by the structure of the following query, with a from-select for each *.

```
from   $Bib |= .bib.book.author[$A]
select author[ authorname[$A]
               | from $Bib |= .bib.book[author[$A]
                                         | $OtherFields
                                        ]
                 select book[$OtherFields]
             ]
```

This query performs a nested loop. For each binding of `$A` to a different author, it returns an edge `author[authorname[$A]]|book[...]|...|book[...]]`, where `book[...]|...|book[...]` is the result of the inner query, i.e. it contains one book element for each book whose author is `$A`. As in a previous example, we extract, from the input book, all the fields but the author.

This query also exemplifies the double role of variables inside formulas. The outer formula provides the bindings for `$A` that satisfy the outer condition, while the inner formula, that is evaluated once for each binding `[$A = C]`, provides those bindings of `$A` and `$OtherFields` that bind `$A` with `C` and satisfy the inner condition. Hence, we may say that the first occurrence *binds* `$A` and the others *verify* that binding. This will be formalized later on.

2.4 Checking Schema Properties and Key Constraints

In this section we show how TQL can be used to check structural properties of semistructured data. When a formula `A` expresses a property, we can check it by running a query like `from Q |= A select success`: this query returns `success` if `A` holds for `Q`, and an empty tree otherwise.

As a first example we consider a query that verifies if the tag `title` is mandatory for `book` elements in the `$Bib` document.

```
from $Bib |= bib[Not .book[Not .title[T]]]
select title_is_mandatory
```

The formula `Not .book[Not .title[T]]` means: it is not the case that there exists a book whose contents do not contain any title, i.e. each book contains a title. TQL actually features an operator `!t[A]` defined as `Not .t[Not A]` which we can directly use, as in the following query. Here `!book.title[T]` is an abbreviation for `!book[.title[T]]`, hence it means: for every book there is a title.

```
from $Bib |= bib[ !book.title[T] ]
select title_is_mandatory
```

The formula `!t[A]` is dual to `.t[A]` in the same sense as $\forall x.A$ is dual to $\exists x.A$, or $\wedge$ is dual to $\vee$. In TQL, every primitive operator has a derived dual; this implies that negation can always be pushed inside any operator, hence we can rewrite any query so that only atomic formulae are negated. Actually, when negation appears in a query, in most cases the TQL optimizer pushes it down to the atomic formulae (tree variables, tree emptyness, comparisons), since negation is quite expensive. This is the reason why, although we claim that unlimited negation is an important feature of TQL, we will see very little explicit use of negation in our examples.

The next query verifies that `title` never appears twice in a field, showing once more how | can be used to analyse horizontal properties.

```
from $Bib |= Not bib[.book[ .title[T] | .title[T] ] ]
select title_never_appears_twice
```

Another important property is whether a given tag is a primary key. There are many possible generalizations of the relational notion of key to the semistructured case. The statement below, for example, says that `title` is a mandatory field, and that you cannot find two separate books with the same title (more precisely, with one title in common).

```
from $Bib |=
   bib[!book[.title[T]]
       And foreach $X. Not (.book.title[$X] |
                            .book.title[$X])
      ]
select each_title_is_key
```

Of course, if the system knows that `$Bib` satisfies `bib[!book[.title[T]]]`, this knowledge implies that

```
bib[!book[.title[T]] And foreach $X. Not (.book.title[$X] | .book.title[$X]) ]
```

is equivalent (over `$Bib`) to

```
bib[foreach $X. Not (.book.title[$X] | .book.title[$X])].
```

Properties like `bib[!book[.title[T]]]` can be easily derived from type declaration expressed using TQL logic (see Section 6.1), and the equivalence above is a simple consequence of the rules that we will present later (Section 4.2). We do not comment further on this point, since this kind of optimization is not exploited by the current implementation of TQL.

The next query checks that the `$Bib` element contains only elements labelled `book`, by asking that each tag inside the outer `bib` is equal to `book`.

```
from $Bib |= bib[foreach $x .$x[T] implies $x=book]
select only_book_inside_bib
```

This query can be rewritten using path operators as follows:

```
from $Bib |= bib[Not (.Not book)[T]]
select only_book_inside_bib
```

Here `Not book` is a tag-expression that stands for any tag different from `book`. Hence, `.Not book[T]` means: there exists a subelement whose tag is different from `book`. Hence, `Not (.Not book[T])` means: there exists no subelement whose tag is different from `book`.

2.5 Extracting the Tags That Satisfy a Property

Every query Q in the previous subsection checks a property P of a tag `t`. In all such cases, if we substitute, in Q, `t` with a tag variable, we obtain a query that finds the set of *all* tags that satisfy P.

For example, we can extract all keys of books by taking the query that checks whether `title` is a key, and substituting `title` with `$k`, as follows:

```
from $Bib |=
     bib[!book[.$k[T]]
         And foreach $X. Not (.book.$k[$X] |
                              .book.$k[$X])
        ]
select key[$k]
```

In the query above we have a universal quantification `foreach $X` of a formula with a free variable `$k`. We are not aware of other query languages where such a quantification is allowed, and the query evaluation algorithm we exploit to support it is quite sophisticated. It is described in [19].

Generalisation by simple substitution is not possible in other query languages, For example, in XQuery, one has to modify the structure of the query, for example by adding an outer `for` clause to bind the variable that replaces that tag.

A similar generalisation can be performed for the queries that check whether a label is mandatory, or occurs only once, inside another one. We present below a query that almost produces a DTD for any input XML file (modulo ordering) by extracting all the tags in the file and listing, for each of them, all the labels that must or may appear, and distinguishing among them the ones which may be repeated and the ones which only appear once. While it may look frightening, it has just been obtained by a trivial generalization of the simple queries we presented above.

We first extract all tags that appear anywhere (`.%*.$tag...`) and contain some subtag (`.%*.$tag[.%[T]]`). For each such tag, we return a structure `$tag[mandatory_subtags*[] | optional_subtags*[] | list_subtags*[] | non_list_subtags*[] ]` that categorizes its subtags.

A subtag is mandatory if it is never the case that we find `$tag` without `$subtag` inside: `Not (.%*.$tag[Not .$subtag[T]])`.

A subtag is optional if there is a `$tag` element with a `$subtag` inside, and there is one with no `$subtag` inside: `.%*.$tag[ .$subtag[T]] And .%*.$tag[Not .$subtag[T]]`.

A subtag is a list-subtag if there is a `$tag` element where it appears twice: `.%*.$tag[ .$subtag[T] | .$subtag[T] ]`.

A subtag is a non-list-subtag if it is sometimes appear once (`.%*.$tag[ .$subtag[T]]`) but it never appears twice: `Not .%*.$tag[ .$subtag[T] | .$subtag[T] ]`.

```
from $parts |= .%*.$tag[.%[T]]
select $tag[ mandatory_subtags
                 [from $parts |= Not (.%*.$tag[Not .$subtag[T]])
                  select $subtag[]
                 ]
             | optional_subtags
                 [from $parts |= .%*.$tag[ .$subtag[T]]
                                 And .%*.$tag[Not .$subtag[T]]
                  select $subtag[]
```

```
              ]
          | list_subtags
              [from $parts |= .%*.$tag[ .$subtag[T] | .$subtag[T] ]
               select $subtag[]
              ]
          | non_list_subtags
              [from $parts |= .%*.$tag[ .$subtag[T]]
                               And Not .%*.$tag[ .$subtag[T] | .$subtag[T] ]
               select $subtag[]
              ]
        ]
```

2.6 Recursion

TQL logic also includes two monotonic recursion operators (`rec` and `maxrec`), very similar to the μ and ν operators (minimal and maximal fix point) of modal logic. These can be used to traverse arbitrarily deep paths, generalizing the `.%*` operator we have seen before, and to express recursive tree properties. Consider for example the following formula:

```
rec $Binary. 0 or (%[$Binary] | %[$Binary])
```

The formula describes a binary tree, defined as either an empty tree, or a tree with two children, both of them binary.

The following query features a combination of horizontal analysis and vertical recursion. In order to check whether the tag `tt` only appears once, we split the source into one edge where `tt` only appears once, and the rest where `tt` never appears. In the first edge, either `tt` appears immediately, and never more (`tt[NoTtHere]`), or it is not here, but appears once inside (`%[$ttOnce] And Not tt[T]`). Hence, the formula looks like:

```
rec $ttOnce. ( tt[NoTtHere] or (%[$ttOnce] And Not tt[T] ))
             | NoTtHere
```

This is the actual query, where `NoTtHere` is expressed by `Not .%*.tt[T]`.

```
from $Source |= rec $ttOnce. ( tt[Not .%*.tt[T]] or (%[$ttOnce] And Not tt[T] ))
                             | Not .%*.tt[T]
select ttAppearsOnce
```

As before, by substituting `tt` with a variable, we get a query that computes the tags that only appear once.

We hope that the reader has now some curiosity about the complete and formal definition of TQL. This is theme of the next sections.

3 TQL Data Model

We represent semistructured data as *information trees*. In this section we first define information trees, then we give a syntax to denote them, and finally we define an equivalence relation that determines when two different expressions denote the same information tree.

3.1 Information Trees

In this section, we formally define unordered edge-labeled trees as nested multisets; of course, any other model for unordered labeled trees would do. Ordered trees could be represented as nested lists. This option would have an impact on the logic, where the symmetric $\mathcal{A} \mid \mathcal{B}$ operator could be replaced by an asymmetric one, $\mathcal{A};\mathcal{B}$. This change might actually simplfy some aspects of the logic, but in this paper we stick to the original notion of unordered trees from [15].

For a given set of *labels* Λ, we define the set $\mathcal{IT}$ of information trees, ranged over by I, as the smallest collection such that:

- the empty multiset, $\{\}$, is in $\mathcal{IT}$; we will use $\mathbf{0}$ as a notation for $\{\}$;
- 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 $\{\langle m, I\rangle\}$;
- $\mathcal{IT}$ is closed under multiset union $\biguplus_{j\in J} M(j)$, where J is an index set, and $M \in J \to \mathcal{IT}$; we will use $Par_{j\in J}\ M(j)$ as a notation for $\biguplus_{j\in J} M(j)$, and $I \mid I'$ for binary union $I \uplus I'$.

3.2 Information Terms

We denote finite information trees by the following syntax of information term (info-terms), borrowed from the ambient calculus [14]. We define a function $[\![F]\!]$ mapping the info-term F to the denoted information tree.

Info-terms and their information tree meaning

$F ::=$ info-term

$\mathbf{0}$	denoting the empty multiset
$m[F]$	denoting the multiset $\{\langle m, F\rangle\}$
$F \mid F$	denoting multiset union

$$
\begin{array}{llllll}
[\![\mathbf{0}]\!] & =_{def} & \mathbf{0} & =_{def} & \{\} \\
[\![m[F]]\!] & =_{def} & m[[\![F]\!]] & =_{def} & \{\langle m, [\![F]\!]\rangle\} \\
[\![F' \mid F'']\!] & =_{def} & [\![F']\!] \mid [\![F'']\!] & =_{def} & [\![F']\!] \uplus [\![F'']\!]
\end{array}
$$

We 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 (integers, strings...), hence $5[\mathbf{0}]$, or 5, is a legitimate info-term. We assume that "|" associates to the right, i.e. $F \mid F' \mid F''$ is read $F \mid (F' \mid F'')$.

3.3 Congruence over Info-Terms

The interpretation of info-terms as information trees induces an equivalence relation $F \equiv F'$ on info-terms. This relation is called *info-term congruence*, and it can be axiomatized as follows.

Congruence over info-terms

$F \equiv F$
$F' \equiv F \Rightarrow F \equiv F'$
$F \equiv F', F' \equiv F'' \Rightarrow F \equiv F''$
$F \equiv F' \Rightarrow m[F] \equiv m[F']$
$F \equiv F' \Rightarrow F \mid F'' \equiv F' \mid F''$

$F \mid \mathbf{0} \equiv F$
$F \mid F' \equiv F' \mid F$
$(F \mid F') \mid F'' \equiv F \mid (F' \mid F'')$

This axiomatization of congruence is sound and complete with respect to the information tree semantics. That is, $F \equiv F'$ if and only if F and F' represent the same information tree.

3.4 Information Trees and other data models

We can compare our information trees with three popular models for semistructured data: OEM data [29], UnQL trees [8], and XML Query Data Model [3]. The first obvious difference is that OEM and UnQL models can be used to represent both trees and graphs, while here we focus only on trees. We are currently working on extending our model to include labeled graphs as well, but we prefer to focus on the simpler issue of trees, which is rich enough to warrant a separate study.

UnQL trees are characterized by the fact that they are considered modulo bisimulation, which essentially means that information trees are seen as sets instead of multisets. For example, $m[n[] \mid n[]]$ is considered the same as $m[n[]]$; hence UnQL trees are more abstract, in the precise sense that they identify more terms than we do.

On the other hand, information trees are more abstract than OEM data, since OEM data can distinguish a DAG from its tree-unfolding.

Our data model is essentially an unordered version of the XML Query Data Model, as defined by the W3C [3]. Apart from order, the other main difference is that the W3C model consider seven different kinds of nodes (elements, attributes, text, ...), while we only consider one (essentially, elements), and the W3C model also assigns a node identity to every node, which we do not consider. In practice, the node identity allows two nodes to be compared in a way that distinguishes them if they have been built by two different applications of a node constructor.

The W3C model describes data as node-labelled forests, while we talk in term of edge-labelled trees. The two are perfectly isomorphic. TQL data can be seen as node-labelled forests by interpreting $\mathbf{0}$ as the empty forest, $F \mid F'$ as forest union, and $t[F]$ as a tree, rooted in a node labelled by t, whose children are the trees in the forest F.

Finally, the implemented version of TQL has a richer data model, since we consider there two types of edges (or "nodes"), element edges $t[F]$ and text (or $PCData$) edges $'t$, which always lead to a leaf. Text edges have very little impact on the language structure, so in this paper we will simply assume that a piece of text t in the XML input is mapped to a terminal edge $t[\mathbf{0}]$.

4 The Tree Logic

In this section we present the tree logic. The tree logic is based on Cardelli and Gordon's modal ambient logic, defined with the aim of specifying spatial and temporal properties of the mobile processes that can be described through the ambient calculus [15]. The ambient logic is particularly attractive because it is equipped with a large set of logical laws for tree-like structures, in particular logical equivalences, that provide a foundation for query rewriting rules and query optimization.

We start here from a subset of the ambient logic as presented in [15], but we enrich it with information tree variables, label comparison, and recursion.

4.1 Formulas

The syntax of the tree logic formulas is presented in the following table.

The symbol $\sim$, in the label comparison clause, stands for any label comparison operator chosen in a predefined family Θ; we will assume that Θ contains at least equality, the SQL string matching operator *like*, and their negations. A recursion variable ξ can only appear positively in its scope; this means that an even number of negations must be traversed in the path that goes from each occurrence of ξ to its binder.

Formulas:

$\eta ::=$	label expression
n	label constant
x	label variable
$\mathcal{A}, \mathcal{B} ::=$	formula
$\mathbf{0}$	empty tree
$\eta[\mathcal{A}]$	location
$\mathcal{A} \mid \mathcal{B}$	composition
$\mathbf{T}$	true
$\neg\mathcal{A}$	negation
$\mathcal{A} \wedge \mathcal{B}$	conjunction
$\mathcal{X}$	tree variable
$\exists x.\mathcal{A}$	quantification over label variables
$\exists \mathcal{X}.\mathcal{A}$	quantification over tree variables
$\eta \sim \eta'$	label comparison
ξ	recursion variable
$\mu\xi.\mathcal{A}$	recursive formula (least fixpoint); ξ may appear only positively

The interpretation of a formula $\mathcal{A}$ is given by a semantic map $[\![\mathcal{A}]\!]_{\rho,\delta}$ that maps $\mathcal{A}$ to a set of information trees, with respect to the valuations ρ and δ. 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 style makes the semantics of recursive formulas easier to define.

Formulas as sets of information trees

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

$[\![\mathbf{0}]\!]_{\rho,\,\delta}$ is the singleton $\{\mathbf{0}\}$. $[\![\eta[\mathcal{A}]]\!]_{\rho,\,\delta}$ contains the information tree $m[I]$, if $m = \rho(\eta)$ and I is in $[\![\mathcal{A}]\!]_{\rho,\,\delta}$. (We assume that ρ maps any label in Λ to itself, so that we can apply ρ to η even when η is not a variable.) For each I in $[\![\mathcal{A}]\!]_{\rho,\,\delta}$ and I' in $[\![\mathcal{B}]\!]_{\rho,\,\delta}$, $[\![\mathcal{A} \mid \mathcal{B}]\!]_{\rho,\,\delta}$ contains the information tree $I \mid I'$. $[\![\mathbf{T}]\!]_{\rho,\,\delta}$ is the set of all information trees (while its negation $\mathbf{F}$ denotes the empty set). $[\![\neg\mathcal{A}]\!]_{\rho,\,\delta}$ is the complement of $[\![\mathcal{A}]\!]_{\rho,\,\delta}$ with respect to the set of all information trees $\mathcal{IT}$. I is in $[\![\mathcal{A} \wedge \mathcal{B}]\!]_{\rho,\,\delta}$ if it is in $[\![\mathcal{A}]\!]_{\rho,\,\delta}$ and in $[\![\mathcal{B}]\!]_{\rho,\,\delta}$. I is in $[\![\exists x.\mathcal{A}]\!]_{\rho,\,\delta}$ if there exists some value n for x such that I is in $[\![\mathcal{A}]\!]_{\rho[x\mapsto n],\,\delta}$. Here $\rho[x \mapsto n]$ denotes the subtitution that maps x to n and otherwise coincides with ρ. $[\![\eta \sim \eta']\!]_{\rho,\,\delta}$ is the set $\mathcal{IT}$ if the comparison holds, else it is the empty set. $[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta}$ is the least fixpoint (with respect to set inclusion) of the monotonic function that maps any set of information trees S to $[\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]}$.

The meaning of a variable $\mathcal{X}$ is given by the valuation ρ. Valuations are the mechanism that connects our logic to pattern matching; for example, $m[n[\mathbf{0}]]$ is in $[\![x[\mathcal{X}]]\!]_{\rho,\,\delta}$ if ρ maps x to m and $\mathcal{X}$ to $n[\mathbf{0}]$. The process of finding all possible ρ's such that $I \in [\![\mathcal{A}]\!]_{\rho,\,\delta}$ is our logic-based way of finding all possible answers to a query with respect to a database I.

We say that F *satisfies* $\mathcal{A}$ *under* ρ, δ, when the information tree $[\![F]\!]$ is in the set $[\![\mathcal{A}]\!]_{\rho,\,\delta}$, and then we write $F \vDash_{\rho,\delta} \mathcal{A}$:

$$F \vDash_{\rho,\delta} \mathcal{A} \;=_{def}\; [\![F]\!] \in [\![\mathcal{A}]\!]_{\rho,\,\delta}$$

Satisfaction enjoys the following properties, which are easily derived and help making the above semantic definition more explicit. These properties form the basis of the matching algorithm of F against $\mathcal{A}$.

Some properties of satisfaction

$F \vDash_{\rho,\delta} \mathbf{0}$	$\Leftrightarrow$	$F \equiv \mathbf{0}$
$F \vDash_{\rho,\delta} \eta[\mathcal{A}]$	$\Leftrightarrow$	$\exists F'.\ F \equiv \rho(\eta)[F'] \ \wedge\ F' \vDash_{\rho,\delta} \mathcal{A}$
$F \vDash_{\rho,\delta} \mathcal{A} \mid \mathcal{B}$	$\Leftrightarrow$	$\exists F', F''.\ F \equiv F' \mid F'' \ \wedge\ F' \vDash_{\rho,\delta} \mathcal{A} \ \wedge\ F'' \vDash_{\rho,\delta} \mathcal{B}$
$F \vDash_{\rho,\delta} \mathbf{T}$		
$F \vDash_{\rho,\delta} \neg\mathcal{A}$	$\Leftrightarrow$	$\neg(F \vDash_{\rho,\delta} \mathcal{A})$
$F \vDash_{\rho,\delta} \mathcal{A} \wedge \mathcal{B}$	$\Leftrightarrow$	$F \vDash_{\rho,\delta} \mathcal{A} \ \wedge\ F \vDash_{\rho,\delta} \mathcal{B}$
$F \vDash_{\rho,\delta} \exists x.\mathcal{A}$	$\Leftrightarrow$	$\exists m \in \Lambda.\ F \vDash_{\rho[x\mapsto m],\delta} \mathcal{A}$
$F \vDash_{\rho,\delta} \exists \mathcal{X}.\mathcal{A}$	$\Leftrightarrow$	$\exists I \in \mathcal{IT}.\ F \vDash_{\rho[\mathcal{X}\mapsto I],\delta} \mathcal{A}$
$F \vDash_{\rho,\delta} \eta \sim \eta'$	$\Leftrightarrow$	$\rho(\eta) \sim \rho(\eta')$
$F \vDash_{\rho,\delta} \mu\xi.\mathcal{A}$	$\Leftrightarrow$	$F \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\}$
$F \vDash_{\rho,\delta} \mathcal{X}$	$\Leftrightarrow$	$[\![F]\!] = \rho(\mathcal{X})$
$F \vDash_{\rho,\delta} \xi$	$\Leftrightarrow$	$[\![F]\!] \in \delta(\xi)$

4.2 Some Derived Formulas

As usual, negation allows us to define many useful derived operators, as described in the following table.

Derived formulas:

$\eta[\Rightarrow \mathcal{A}]$	$=_{def}$	$\neg(\eta[\neg\mathcal{A}])$	$\mathcal{A} \mid\mid \mathcal{B}$	$=_{def}$	$\neg(\neg\mathcal{A} \mid \neg\mathcal{B})$
$\mathbf{F}$	$=_{def}$	$\neg\mathbf{T}$	$\mathcal{A} \vee \mathcal{B}$	$=_{def}$	$\neg(\neg\mathcal{A} \wedge \neg\mathcal{B})$
$\forall x.\mathcal{A}$	$=_{def}$	$\neg(\exists x.\neg\mathcal{A})$	$\forall \mathcal{X}.\mathcal{A}$	$=_{def}$	$\neg(\exists \mathcal{X}.\neg\mathcal{A})$
$\nu\xi.\mathcal{A}$	$=_{def}$	$\neg(\mu\xi.\neg\mathcal{A}\{\xi \leftarrow \neg\xi\})$			

$F \vDash m[\Rightarrow \mathcal{A}]$ means that 'it is not true that, for some F', $F \equiv m[F']$ and not $F' \vDash \mathcal{A}$', i.e. 'if F has the shape $m[F']$, then $F' \vDash \mathcal{A}$'. To appreciate the difference between $m[\mathcal{A}]$ and its dual $m[\Rightarrow \mathcal{A}]$, consider the following statements.

- F is a book where *Date* is an author: $F \vDash book[.author[Date]]$
- If F is a book, then *Date* is an author: $F \vDash book[\Rightarrow .author[Date]]$

$F \vDash \mathcal{A} \mid\mid \mathcal{B}$ means that 'it is not true that, for some F' and F'', $F \equiv F' \mid F''$ and $F' \vDash \neg\mathcal{A}$ and $F'' \vDash \neg\mathcal{B}$', which means: for *every* decomposition of F into $F' \mid F''$, *either* $F' \vDash \mathcal{A}$ *or* $F'' \vDash \mathcal{B}$. To appreciate the difference between the $\mid$ and the $\mid\mid$ operators, consider the following statements.

- There exists a decomposition of F into F' and F'', such that F' satisfies $book[\mathcal{A}]$, and F'' satsfies $\mathbf{T}$; i.e., there is a book inside F that satisfies $\mathcal{A}$: $F \vDash book[\mathcal{A}] \mid \mathbf{T}$
- For every decomposition of F into F' and F'', either F' satisfies $book[\Rightarrow \mathcal{A}]$, or F'' satisfies $\mathbf{F}$; i.e., every book inside F satisfies $\mathcal{A}$: $F \vDash book[\Rightarrow \mathcal{A}] \mid\mid \mathbf{F}$

The dual of the least fixpoint operator $\mu\xi.\mathcal{A}$ is the greatest fixpoint operator $\nu\xi.\mathcal{A}$. For example $\mu\xi.\xi$ is equivalent to $\mathbf{F}$, while $\nu\xi.\xi$ is equivalent to $\mathbf{T}$. More interestingly, $\mu\xi.\mathbf{0} \vee m[\xi]$ describes every information tree that matches $m[m[\ldots m[]]]$, and, on finite trees, it is equivalent to $\nu\xi.\mathbf{0} \vee m[\xi]$. However, if we consider infinite trees, the distinction between least and greatest fixpoint becomes more important. For example, the infinite tree $m[m[\ldots]]$ satisfies $\nu\xi.\mathbf{0} \vee m[\xi]$, but does not satisfy $\mu\xi.\mathbf{0} \vee m[\xi]$. When we consider only finite trees, the μ and ν operators are quite similar in practice, since most interesting formulas have a single fixpoint.

Satisfaction over the derived operators enjoys the following properties, most of which are easily derived from the definition, while others are more subtle. For example, the properties of greatest fixpoints include a coinduction principle. Again, these properties form the basis for a matching algorithm.

Some properties of satisfaction for derived formulas

$$
\begin{array}{lll}
\neg F \vDash_{\rho,\delta} \mathbf{F} & & \\
F \vDash_{\rho,\delta} \eta[\Rightarrow \mathcal{A}] & \Leftrightarrow & \forall F'.\ (F \equiv \rho(\eta)[F'] \ \Rightarrow F' \vDash_{\rho,\delta} \mathcal{A}) \\
F \vDash_{\rho,\delta} \mathcal{A} \mid\mid \mathcal{B} & \Leftrightarrow & \forall F', F''.\ F \equiv F' \mid F'' \ \Rightarrow \ (F' \vDash_{\rho,\delta} \mathcal{A} \ \vee \ F'' \vDash_{\rho,\delta} \mathcal{B}) \\
F \vDash_{\rho,\delta} \mathcal{A} \vee \mathcal{B} & \Leftrightarrow & F \vDash_{\rho,\delta} \mathcal{A} \ \vee \ F \vDash_{\rho,\delta} \mathcal{B} \\
F \vDash_{\rho,\delta} \forall x.\mathcal{A} & \Leftrightarrow & \forall m \in \Lambda.\ F \vDash_{\rho[x \mapsto m],\delta} \mathcal{A} \\
F \vDash_{\rho,\delta} \forall \mathcal{X}.\mathcal{A} & \Leftrightarrow & \forall I \in \mathcal{IT}.\ F \vDash_{\rho[\mathcal{X} \mapsto I],\delta} \mathcal{A} \\
F \vDash_{\rho,\delta} \nu\xi.\mathcal{A} & \Leftrightarrow & F \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \nu\xi.\mathcal{A}\} \\
F \vDash_{\rho,\delta} \nu\xi.\mathcal{A} & \Leftrightarrow & \exists \mathcal{B}.\ F \vDash_{\rho,\delta} \mathcal{B} \ \wedge \ \forall F'.\ F' \vDash_{\rho,\delta} \mathcal{B} \Rightarrow F' \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \mathcal{B}\}
\end{array}
$$

Many logical equivalences have been derived for the ambient logic, and are inherited by the tree logic. We list some of them here. These equivalences can be exploited by a query logical optimizer.

Some equations

$$
\begin{array}{ll}
\eta[\mathcal{A}] \wedge \mathbf{0} \Leftrightarrow \mathbf{F} & \eta[\Rightarrow \mathcal{A}] \vee \neg\mathbf{0} \Leftrightarrow \mathbf{T} \\
\eta[\mathcal{A}] \wedge \eta'[\mathcal{A}'] \Leftrightarrow \eta[\mathcal{A} \wedge \mathcal{A}'] \wedge \eta = \eta' & \eta[\Rightarrow \mathcal{A}] \vee \eta'[\Rightarrow \mathcal{A}'] \Leftrightarrow \eta[\Rightarrow \mathcal{A} \vee \mathcal{A}'] \vee \eta \neq \eta' \\
\eta[\mathcal{A}] \wedge (\eta'[\mathcal{A}'] \mid \eta''[\mathcal{A}''] \mid \mathcal{A}''') \Leftrightarrow \mathbf{F} & \eta[\Rightarrow \mathcal{A}] \vee (\eta'[\Rightarrow \mathcal{A}'] \mid\mid \eta'[\Rightarrow \mathcal{A}'] \mid\mid \mathcal{A}''') \Leftrightarrow \mathbf{T}
\end{array}
$$

$\eta[\mathcal{A}]$	$\Leftrightarrow \eta[\mathbf{T}] \;\wedge\; \eta[\Rightarrow \mathcal{A}]$	$\eta[\Rightarrow \mathcal{A}]$	$\Leftrightarrow \eta[\mathbf{T}] \;\Rightarrow\; \eta[\mathcal{A}]$
$\eta[\mathbf{F}]$	$\Leftrightarrow \mathbf{F}$	$\eta[\Rightarrow \mathbf{T}]$	$\Leftrightarrow \mathbf{T}$
$\eta[\mathcal{A} \wedge \mathcal{A}']$	$\Leftrightarrow \eta[\mathcal{A}] \wedge \eta[\mathcal{A}']$	$\eta[\Rightarrow \mathcal{A} \vee \mathcal{A}']$	$\Leftrightarrow \eta[\Rightarrow \mathcal{A}] \vee \eta[\Rightarrow \mathcal{A}']$
$\eta[\mathcal{A} \vee \mathcal{A}']$	$\Leftrightarrow \eta[\mathcal{A}] \vee \eta[\mathcal{A}']$	$\eta[\Rightarrow \mathcal{A} \wedge \mathcal{A}']$	$\Leftrightarrow \eta[\Rightarrow \mathcal{A}] \wedge \eta[\Rightarrow \mathcal{A}']$
$\eta[\exists x.\mathcal{A}]$	$\Leftrightarrow \exists x.\eta[\mathcal{A}] \ (x \neq \eta)$	$\eta[\Rightarrow \forall x.\mathcal{A}]$	$\Leftrightarrow \forall x.\eta[\Rightarrow \mathcal{A}] \ (x \neq \eta)$
$\eta[\forall x.\mathcal{A}]$	$\Leftrightarrow \forall x.\eta[\mathcal{A}] \ (x \neq \eta)$	$\eta[\Rightarrow \exists x.\mathcal{A}]$	$\Leftrightarrow \exists x.\eta[\Rightarrow \mathcal{A}] \ (x \neq \eta)$
$\eta[\exists \mathcal{X}.\mathcal{A}]$	$\Leftrightarrow \exists \mathcal{X}.\eta[\mathcal{A}]$	$\eta[\Rightarrow \forall \mathcal{X}.\mathcal{A}]$	$\Leftrightarrow \forall \mathcal{X}.\eta[\Rightarrow \mathcal{A}]$
$\eta[\forall \mathcal{X}.\mathcal{A}]$	$\Leftrightarrow \forall \mathcal{X}.\eta[\mathcal{A}]$	$\eta[\Rightarrow \exists \mathcal{X}.\mathcal{A}]$	$\Leftrightarrow \exists \mathcal{X}.\eta[\Rightarrow \mathcal{A}]$
$\mathcal{A} \mid \mathbf{0}$	$\Leftrightarrow \mathcal{A}$	$\mathcal{A} \parallel \neg\mathbf{0}$	$\Leftrightarrow \mathcal{A}$
$\mathcal{A} \mid \mathcal{A}'$	$\Leftrightarrow \mathcal{A}' \mid \mathcal{A}$	$\mathcal{A} \parallel \mathcal{A}'$	$\Leftrightarrow \mathcal{A}' \parallel \mathcal{A}$
$(\mathcal{A} \mid \mathcal{A}') \mid \mathcal{A}''$	$\Leftrightarrow \mathcal{A} \mid (\mathcal{A}' \mid \mathcal{A}'')$	$(\mathcal{A} \parallel \mathcal{A}') \parallel \mathcal{A}''$	$\Leftrightarrow \mathcal{A} \parallel (\mathcal{A}' \parallel \mathcal{A}'')$
$\mathbf{T} \mid \mathbf{T}$	$\Leftrightarrow \mathbf{T}$	$\mathbf{F} \parallel \mathbf{F}$	$\Leftrightarrow \mathbf{F}$
$\mathcal{A} \mid \mathbf{F}$	$\Leftrightarrow \mathbf{F}$	$\mathcal{A} \parallel \mathbf{T}$	$\Leftrightarrow \mathbf{T}$
$\mathcal{A} \mid (\mathcal{A}' \vee \mathcal{A}'')$	$\Leftrightarrow (\mathcal{A} \mid \mathcal{A}') \vee (\mathcal{A} \mid \mathcal{A}'')$	$\mathcal{A} \parallel (\mathcal{A}' \wedge \mathcal{A}'')$	$\Leftrightarrow (\mathcal{A} \parallel \mathcal{A}') \wedge (\mathcal{A} \parallel \mathcal{A}'')$
$\mathcal{A} \mid \exists x.\mathcal{A}'$	$\Leftrightarrow \exists x.\mathcal{A} \mid \mathcal{A}' \ (x \notin FV(\mathcal{A}))$	$\mathcal{A} \parallel \forall x.\mathcal{A}'$	$\Leftrightarrow \forall x.\mathcal{A} \parallel \mathcal{A}' \ (x \notin FV(\mathcal{A}))$
$\mathcal{A} \mid \exists \mathcal{X}.\mathcal{A}'$	$\Leftrightarrow \exists \mathcal{X}.\mathcal{A} \mid \mathcal{A}' \ (\mathcal{X} \notin FV(\mathcal{A}))$	$\mathcal{A} \parallel \forall \mathcal{X}.\mathcal{A}'$	$\Leftrightarrow \forall \mathcal{X}.\mathcal{A} \parallel \mathcal{A}' \ (\mathcal{X} \notin FV(\mathcal{A}))$

4.3 Path Formulas

All query languages for semistructured data provide some way of retrieving all data that is reachable through a *path* described by a regular expression. The tree logic is powerful enough to express this kind of queries. We show this fact here by defining a syntax for path expressions, and showing how these expressions can be translated into the logic. This way, we obtain a more compact and readable way of expressing common queries, as partially exemplified in Section 2.

Consider the following statement: $\mathcal{X}$ is some book found in the $BOOKS$ collection, and some author of $\mathcal{X}$ is $Abiteboul$. We can express it in the logic using the $m[\mathcal{A}] \mid \mathbf{T}$ pattern as:

$$BOOKS \vDash book[\mathcal{X} \wedge (author[Abiteboul] \mid \mathbf{T})] \mid \mathbf{T}$$

Using the special syntax of path expressions, we express the same condition as follows.

$$BOOKS \vDash .book(\mathcal{X}).author[Abiteboul]$$

Our path expressions support also the following features:

- Universally quantified paths: $\mathcal{X}$ is a book and *every* author of $\mathcal{X}$ is Abiteboul.

$$BOOKS \vDash .book(\mathcal{X})!author[Abiteboul]$$

- Label negation: $\mathcal{X}$ is a book where $Date$ is the value of a field, but is not the author.

$$BOOKS \vDash .book(\mathcal{X}).(\neg author)[Date]$$

- Path disjunction: $\mathcal{X}$ is a book that either deals with SSD or cites some book $\mathcal{Y}$ that only deals with SSD.

$$BOOKS \vDash .book(\mathcal{X})(.keyword \vee .cites.book(\mathcal{Y})!keyword)[SSD]$$

- Path iteration (Kleene star): $\mathcal{X}$ is a book that either deals with SSD, or from which you can reach, through a chain of citations, a book that deals with SSD.

$$BOOKS \vDash .book(\mathcal{X})(.cites.book)^*.keyword[SSD]$$

- Label matching: there exists a path through which you can reach some field $\mathcal{X}$ whose label contains e and $mail$ (% matches any substring).

$$BOOKS \vDash (.\%)^*(.\%e\%mail\%)[\mathcal{X}]$$

We now define the syntax of paths and its interpretation.

Path formulas:

$\alpha ::=$	label matching expression	
η		matches any n such that n *like* η
$\neg\alpha$		matches whatever α does not match
$\beta ::=$	path element	
$.\alpha$		some edge matches α
$!\alpha$		each edge matches α
$p, q ::=$	path	
β		elementary path
pq		path concatenation
p^*		Kleene star
$p \vee q$		disjunction
$p(\mathcal{X})$		naming the tree at the end of the path

A path-based formula $p[\mathcal{A}]$ can be translated into the tree logic as shown below.
We first define the tree formula $Matches(x, \alpha)$ as follows:

$$\begin{array}{lll} Matches(x,\eta) & =_{def} & x \text{ like } \eta \\ Matches(x,\neg\alpha) & =_{def} & \neg Matches(x,\alpha) \end{array}$$

Path elements are interpreted by a translation, $[\![\_]\!]^p$, into the logic, using the patterns $m[\mathcal{A}] \mid \mathbf{T}$ and $m[\Rightarrow \mathcal{A}] \mid\mid \mathbf{F}$ that we have previously presented:

$$\begin{array}{lll} [\![.\alpha[\mathcal{A}]]\!]^p & =_{def} & (\exists x.Matches(x,\alpha) \ \wedge \ x[[\![\mathcal{A}]\!]^p]) \mid \mathbf{T} \\ [\![!\alpha[\mathcal{A}]]\!]^p & =_{def} & (\forall x.Matches(x,\alpha) \ \Rightarrow \ x[\Rightarrow [\![\mathcal{A}]\!]^p]) \mid\mid \mathbf{F} \end{array}$$

General paths are interpreted as follows. $p^*[\mathcal{A}]$ is recursively interpreted as 'either $\mathcal{A}$ holds here, or $p^*[\mathcal{A}]$ holds after traversing p'. Target naming $p(\mathcal{X})[\mathcal{A}]$ means: at the end of p you find $\mathcal{X}$, and $\mathcal{X}$ satisfies $\mathcal{A}$; hence it is interpreted using logical conjunction. Formally, path interpretation is defined as shown below; path interpretation translates all non-path operators as themselves, as exemplified for $\mathbf{T}$ and $\mid$.

$$\begin{array}{lllllll} [\![pq[\mathcal{A}]]\!]^p & =_{def} & [\![p[q[\mathcal{A}]]]\!]^p & \quad & [\![p^*[\mathcal{A}]]\!]^p & =_{def} & \mu\xi.\mathcal{A} \ \vee \ [\![p[\xi]]\!]^p \\ [\![(p \vee q)[\mathcal{A}]]\!]^p & =_{def} & [\![p[\mathcal{A}]]\!]^p \ \vee \ [\![q[\mathcal{A}]]\!]^p & & [\![p(\mathcal{X})[\mathcal{A}]]\!]^p & =_{def} & [\![p[\mathcal{X} \ \wedge \ \mathcal{A}]]\!]^p \\ [\![\mathbf{T}]\!]^p & =_{def} & \mathbf{T} & & [\![\mathcal{A} \mid \mathcal{A}']\!]^p & =_{def} & [\![\mathcal{A}]\!]^p \mid [\![\mathcal{A}']\!]^p \end{array}$$

5 The Tree Query Language

In this section we build a full query language on top of the logic we have defined.

5.1 The Query Language

A query language must provide the following functionalities:

- binding and selection: a mechanism to select values from the database and to bind them to variables;
- construction of the result: a mechanism to build a result starting from the bindings collected during the previous stage.

Our Tree Query Language (TQL) uses the tree logic for binding and selection, and tree building operations to construct the result. Logical formulas $\mathcal{A}$ are as previously defined.

TQL queries:

$Q ::=$	query
$\quad$ *from* $Q \vDash \mathcal{A}$ *select* Q'	valuation-collecting query
$\quad \mathcal{X}$	matching variable
$\quad \mathbf{0}$	empty result
$\quad Q \mid Q$	composition of results
$\quad \eta[Q]$	nesting of result
$\quad f(Q)$	tree function, for any f in a fixed set Φ

We allow some tree functions f, chosen from a set Φ of functions of type $\mathcal{IT} \rightarrow \mathcal{IT}$, to appear in the query. For example:

- *count(I)*, which yields a tree $n[\mathbf{0}]$, where n is the cardinality of the multiset I;
- *op(I)*, where *op* is a commutative, associative integer function with a neutral element; if all the pairs in I have a shape $n[I']$, where n is a natural number, then *op(I)* combines all the n's using the *op* operation obtaining the integer r, and returns $r[\mathbf{0}]$.

In the implemented systems, the set Φ can be extended by the user with any Java function with the right signature.

The definition of *free variables* in a query is standard, except for the *from* $Q \vDash \mathcal{A}$ *select* Q' from-where case. The binder $Q \vDash \mathcal{A}$ computes valuations for all the variables that are free in $\mathcal{A}$ and uses them to evaluate Q', hence it acts as a binders for all variables that are free in $\mathcal{A}$; this explain the first case in the following table.

Free variables in TQL queries:

$FV(\textit{from } Q \vDash \mathcal{A} \textit{ select } Q')$	$=_{def}$	$FV(Q) \cup (FV(Q') \setminus FV(\mathcal{A}))$
$FV(\mathcal{X})$	$=_{def}$	$\{\mathcal{X}\}$
$FV(\mathbf{0})$	$=_{def}$	$\{\}$
$FV(Q \mid Q')$	$=_{def}$	$FV(Q) \cup FV(Q')$
$FV(\eta[Q])$	$=_{def}$	$FV(\eta) \cup FV(Q)$
$FV(f[Q])$	$=_{def}$	$FV(Q)$

from $Q \vDash \mathcal{A}$, $Q' \vDash \mathcal{A}'$ *select* Q'' is an abbreviation for *from* $Q \vDash \mathcal{A}$ *select from* $Q' \vDash \mathcal{A}'$ *select* Q''.

5.2 Query Semantics

Hereafter $\mathbf{V}$ ranges over finite sets of variables $\mathcal{V}_1, ..., \mathcal{V}_n$, where each variable $\mathcal{V}_i$ is either an information tree variable $\mathcal{X}$, whose universe $U(\mathcal{X})$ is defined to be the set $\mathcal{IT}$ of all information trees, or a label variable x, whose universe $U(x)$ is defined to be the set Λ of all labels. $\rho^{\mathbf{V}}$ ranges over valuations with schema $\mathbf{V}$, i.e. functions mapping each $\mathcal{V}_i$ to an element of $U(\mathcal{V}_i)$.

The semantics of a query is defined with respect to an 'input valuation' $\rho^{\mathbf{V}}$ that binds all the variables that occur free in the query.

The input valuation is used to bind some top-level names, like `$Bib` in Section 2. In a query *from* $Q \vDash \mathcal{A}$ *select* Q', the input valuation of Q' also conveys the valuations (one at a time) extracted by the binder $Q \vDash \mathcal{A}$.

The semantics of a query is defined in the following table. The interesting case is the one for *from* $Q \vDash \mathcal{A}$ *select* Q'. In this case, the subquery Q' is evaluated once for each valuation ρ' that extends the input valuation ρ and such that $[\![Q]\!]_\rho \in [\![\mathcal{A}]\!]_{\rho',\epsilon}$; all the resulting trees $[\![Q']\!]_{\rho'}$ are then combined using Par, the n-ary version of the binary operator $|$, defined in Section 3.

The notation $\rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}}$ means that $\mathbf{V}' \supseteq \mathbf{V}$ and that $\rho'^{\mathbf{V}'}$ and $\rho^{\mathbf{V}}$ coincide over $\mathbf{V}$. This means that $Q \vDash \mathcal{A}$ is restricted to return valuations that extend the input valuation $\rho^{\mathbf{V}}$.

Query semantics

$$
\begin{array}{ll}
[\![\mathcal{X}]\!]_{\rho^{\mathbf{V}}} & = \rho^{\mathbf{V}}(\mathcal{X}) \\
[\![\mathbf{0}]\!]_{\rho^{\mathbf{V}}} & = \mathbf{0} \\
[\![Q \mid Q']\!]_{\rho^{\mathbf{V}}} & = [\![Q]\!]_{\rho^{\mathbf{V}}} \mid [\![Q']\!]_{\rho^{\mathbf{V}}} \\
[\![m[Q]]\!]_{\rho^{\mathbf{V}}} & = m[[\![Q]\!]_{\rho^{\mathbf{V}}}] \\
[\![x[Q]]\!]_{\rho^{\mathbf{V}}} & = \rho^{\mathbf{V}}(x)[[\![Q]\!]_{\rho^{\mathbf{V}}}] \\
[\![f(Q)]\!]_{\rho^{\mathbf{V}}} & = f([\![Q]\!]_{\rho^{\mathbf{V}}}) \\
[\![\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q']\!]_{\rho^{\mathbf{V}}} & = Par_{\rho'^{\mathbf{V}'} \in \{\rho'^{\mathbf{V}'} \ \mid\ \mathbf{V}' = \mathbf{V} \cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ [\![Q]\!]_{\rho^{\mathbf{V}}} \in [\![\mathcal{A}]\!]_{\rho'^{\mathbf{V}'},\epsilon}\}} [\![Q']\!]_{\rho'^{\mathbf{V}'}}
\end{array}
$$

According to this interpretation, the result of a query *from* $Q \vDash \mathcal{A}$ *select* Q' can be an infinite multiset. Therefore, in a nested query, the database Q can be infinite, even if we start from a finite initial database. Obviously, one would not like this to happen in practice. One possible solution is to syntactically restrict Q to a variable $\mathcal{X}$. Another solution is to have a static or dynamic check on the finiteness of the result; one such option is dicussed in Section 5.4.

5.3 Safe Queries

It is well-known that disjunction, negation, and universal quantification create 'safety' problems in logic-based query languages. The same problems appear in our query language.

Consider for example the following query:

$$
\begin{array}{ll}
\mathit{from} & DB \vDash (\mathit{author}[\mathcal{X}] \vee \mathit{autore}[\mathcal{Y}]) \mid \mathbf{T} \\
\mathit{select} & \mathit{author}[\mathcal{X}] \mid \mathit{autore}[\mathcal{Y}]
\end{array}
$$

Intuitively, every entry in DB that is an *author* binds $\mathcal{X}$ but not $\mathcal{Y}$, and vice-versa for *autore* entries. Formally, both situations generate an infinite amount of valuations; for example, if

$\rho(DB) = author[m[]]$, then $\{\rho' \mid [\![DB]\!]_\rho \in [\![\mathcal{A}]\!]_{\rho',\epsilon}\}$ is the infinite set of triples:

$$\{(DB \mapsto author[m[]],\ \mathcal{X} \mapsto m[],\ \mathcal{Y} \mapsto I) \mid I \in \mathcal{IT}\}\ .$$

Negation creates a similar problem. Consider the following query.

$$\begin{array}{ll} \textit{from} & DB \vDash \neg author[\mathcal{X}] \\ \textit{select} & notauthor[\mathcal{X}] \end{array}$$

Its binder, with respect to the above input valuation, generates the following infinite set of bindings:

$$\{(DB \mapsto author[m[]],\ \mathcal{X} \mapsto I) \mid I \in \mathcal{IT}, I \neq m[]\}\}\ ,$$

and the query has the following infinite result:

$$Par_{I \in \mathcal{IT}, I \neq m[]}\ notauthor[I]$$

These queries present two different, but related, problems:

- their semantics depends on the sets Λ and $\mathcal{IT}$ of all possible labels and information trees;
- their semantics is infinite.

We say that a query is safe when its semantics is finite. We say that a formula $\mathcal{A}$ is safe with respect to a set of bound variables $\mathbf{V}$ when, for each information tree I and for each valution $\rho^{\mathbf{V}}$ for $\mathbf{V}$, the set of valuations extracted by $\mathcal{A}$ ($\{\rho'^{\mathbf{V}'} \mid \mathbf{V}' = \mathbf{V} \cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ I \in [\![\mathcal{A}]\!]_{\rho'^{\mathbf{V}'},\epsilon}\}$) is finite; in this case, we also say that, if the variables in $\mathbf{V}$ are bound, then $\mathcal{A}$ *binds* its other free variables. Unsafe formulas are difficult to evaluate since they return an infinite set. But safe formulas can be problematic too, since a safe formula may in general contain an unsafe subformula, as in $\neg autore[\mathcal{X}] \wedge author[\mathcal{X}]$, where the formula is safe but the first conjunct is not. To evaluate such formulas, one can define:

- either a static analysis algorithm that finds an evaluation order for the subformulas, so that the subformula that binds the variables (i.e., produces a finite set of bindings) is evaluated before the unsafe conjunct; when the algorithm cannot found such an order, the query is statically refused as 'not statically safe';
- or an evaluation mechanism that copes with infinite sets of bindings.

Unfortunately, we believe that the first approach cannot be made complete, since query safety is known to be undecidable for the relational tuple calculus [5], and we suspect it is undecidable for our calculus too. Still, such a static analysis can give good results in many situations, hence is an interesting research topic.

For this reason we have studied both approaches. We have defined a formula evaluation mechanism that copes with infinite results, described in Section 7, and a static analysis algorithm which is not complete but can be used to compute a quite precise subset of the variables that are bound by a formula, and is described in Section 5.4.

5.4 Restricted queries

We define here a simple static analysis algorithm to compute a subset of the variables that are bound by a formula, and use it to define a notion of *restricted queries* such that every restricted query is statically guaranteed to be safe, i.e. to generate a finite answer. For simplicity, we do not consider here recursive operators.

We define two predicates $\mathbf{V} \vdash \mathcal{A}^{\mathcal{V}}$ ($\mathcal{A}$ binds $\mathcal{V}$ if $\mathbf{V}$ is bound) and $\mathbf{V} \vdash \mathcal{A}_{\mathcal{V}}$ ($\mathcal{A}$ co-binds $\mathcal{V}$ if $\mathbf{V}$ is bound); the first implies, informally, that for any I and for any valuation for $\mathbf{V}$, $\mathcal{A}$ can only extract finitely many matches from I; the second says that $\neg\mathcal{A}$ binds $\mathcal{V}$ (if $\mathbf{V}$ is bound) (see Property 1 below for a formal definition). The set of pre-bound variables $\mathbf{V}$ is only useful to deal with the equality operator.

The binding predicate is defined as follows. The second column of the table deals with dual operators. We omit the simmetric rules ($y = x$, $n = x$) for the equality case.

The binding predicate

$\mathbf{V} \vdash \neg\mathcal{A}^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}}$		
$\mathbf{V} \vdash \neg\mathcal{A}_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}}$		
$\mathbf{V} \vdash \mathbf{T}_{\mathcal{V}}$ for any $\mathcal{V}$		$\mathbf{V} \vdash \mathbf{F}^{\mathcal{V}}$ for any $\mathcal{V}$	
$\mathbf{V} \vdash \mathcal{X}^{\mathcal{X}}$		$\mathbf{V} \vdash (\neg\mathcal{X})_{\mathcal{X}}$	
$\mathbf{V} \vdash (x = n)^x$		$\mathbf{V} \vdash (x \neq n)_x$	
$\mathbf{V} \vdash (x = y)^x$	$\Leftrightarrow y \in \mathbf{V}$	$\mathbf{V} \vdash (x \neq y)_x$	$\Leftrightarrow y \in \mathbf{V}$
$\mathbf{V} \vdash (\mathcal{A} \wedge \mathcal{B})^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \vee \mathbf{V} \vdash \mathcal{B}^{\mathcal{V}}$	$\mathbf{V} \vdash (\mathcal{A} \vee \mathcal{B})_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \vee \mathbf{V} \vdash \mathcal{B}_{\mathcal{V}}$
$\mathbf{V} \vdash (\mathcal{A} \wedge \mathcal{B})_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \wedge \mathbf{V} \vdash \mathcal{B}_{\mathcal{V}}$	$\mathbf{V} \vdash (\mathcal{A} \vee \mathcal{B})^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \wedge \mathbf{V} \vdash \mathcal{B}^{\mathcal{V}}$
$\mathbf{V} \vdash \eta[\mathcal{A}]^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \vee \eta = \mathcal{V}$	$\mathbf{V} \vdash \eta[\Rightarrow \mathcal{A}]_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \vee \eta = \mathcal{V}$
$\mathbf{V} \vdash (\mathcal{A} \mid \mathcal{B})^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \vee \mathbf{V} \vdash \mathcal{B}^{\mathcal{V}}$	$\mathbf{V} \vdash (\mathcal{A} \mid\mid \mathcal{B})_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \vee \mathbf{V} \vdash \mathcal{B}_{\mathcal{V}}$
$\mathbf{V} \vdash (\mathcal{A} \mid \mathcal{B})_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \wedge \mathbf{V} \vdash \mathcal{B}_{\mathcal{V}}$	$\mathbf{V} \vdash (\mathcal{A} \mid\mid \mathcal{B})^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \wedge \mathbf{V} \vdash \mathcal{B}^{\mathcal{V}}$
$\mathbf{V} \vdash (\forall\mathcal{V}'.\ \mathcal{A})^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \wedge \mathcal{V}' \neq \mathcal{V}$	$\mathbf{V} \vdash (\exists\mathcal{V}'.\ \mathcal{A})_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \wedge \mathcal{V}' \neq \mathcal{V}$
$\mathbf{V} \vdash (\forall\mathcal{V}'.\ \mathcal{A})_{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} \wedge \mathcal{V}' \neq \mathcal{V}$	$\mathbf{V} \vdash (\exists\mathcal{V}'.\ \mathcal{A})^{\mathcal{V}}$	$\Leftrightarrow \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} \wedge \mathcal{V}' \neq \mathcal{V}$

Now we can talk about query safety. A query Q is safe with respect to an input valuation $\rho^{\mathbf{V}}$ if $\mathit{safe}(Q, \mathbf{V})$ holds. A query *from* $Q \vDash \mathcal{A}$ *select* Q' is safe if: all variables in $\mathcal{A}$ are bound, it is safe to evaluate Q, and it is safe to evaluate Q' using the valuations produced by $\mathcal{A}$. Any other query is safe unless it contains an unsafe subquery.

Query safety with respect to an input substitution that binds V

$\mathit{binds}(\mathbf{V}, \mathcal{A}) =_{def} \mathbf{V} \cup \{\mathcal{V} \mid \mathbf{V} \vdash \mathcal{A}^{\mathcal{V}}\}$

$\mathit{safe}(\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q', \mathbf{V}) \Leftrightarrow FV(\mathcal{A}) \subseteq \mathit{binds}(\mathbf{V}, \mathcal{A}) \wedge \mathit{safe}(Q, \mathbf{V}) \wedge \mathit{safe}(Q', \mathit{binds}(\mathbf{V}, \mathcal{A}))$

$\mathit{safe}(\mathcal{X}, \mathbf{V})$	
$\mathit{safe}(\mathbf{0}, \mathbf{V})$	
$\mathit{safe}(Q \mid Q', \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V}) \wedge \mathit{safe}(Q', \mathbf{V})$
$\mathit{safe}(m[Q], \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V})$
$\mathit{safe}(x[Q], \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V})$
$\mathit{safe}(f(Q), \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V})$

The soundness of this analysis is expressed by the following properties.

Property 1 *If I ranges over finite information trees, and $\rho^{\mathbf{V}}$ ranges over valuations mapping the variables in* $\mathbf{V}$ *to labels or to* finite *trees, then:*

$$\begin{array}{lll}
\mathbf{V} \vdash \mathcal{A}^{\mathcal{V}} & \Rightarrow & \forall I,\, \rho^{\mathbf{V}}. \\
& & \textit{the set } \{\rho'^{\mathbf{V}'}(\mathcal{V}) \mid \mathbf{V}' = \mathbf{V} \cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ I \in [\![\mathcal{A}]\!]_{\rho'^{\mathbf{V}'},\epsilon}\} \textit{ is finite} \\
\mathbf{V} \vdash \mathcal{A}_{\mathcal{V}} & \Rightarrow & \forall I,\, \rho^{\mathbf{V}}. \\
& & \textit{the set } \{\rho'^{\mathbf{V}'}(\mathcal{V}) \mid \mathbf{V}' = \mathbf{V} \cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ I \in [\![\neg\mathcal{A}]\!]_{\rho'^{\mathbf{V}'},\epsilon}\} \textit{ is finite}
\end{array}$$

Property 2 *If $\rho^{\mathbf{V}}$ ranges over valuations mapping the variables in* $\mathbf{V}$ *to labels or to* finite *trees, then:*

$$\textit{safe}(Q, \mathbf{V}) \quad \Rightarrow \quad \forall \rho^{\mathbf{V}}.\ [\![Q]\!]_{\rho^{\mathbf{V}}} \textit{is finite}$$

6 Schemas and Constraints

6.1 Schemas

Traditional path-based query languages explore the vertical structure of trees. Our logic can also express easily horizontal structure, as is common in schemas for semistructured data. (E.g. in XML DTDs, XDuce Types [26], and XSD Schemas [2]; however, the present version of our logic only considers unordered structures.)

For example, we can extract the following regular-expression-like sublanguage, inspired by XDuce and XSD types. Every expression of this language denotes a set of information trees:

$\mathbf{0}$	the empty tree
$\mathcal{A} \mid \mathcal{B}$	an $\mathcal{A}$ next to a $\mathcal{B}$
$\mathcal{A} \vee \mathcal{B}$	either an $\mathcal{A}$ or a $\mathcal{B}$
$n[\mathcal{A}]$	an edge n leading to an $\mathcal{A}$
$\mathcal{A}^* =_{def} \mu\xi.\ \mathbf{0} \vee (\mathcal{A} \mid \xi)$	a finite multiset of zero or more $\mathcal{A}$'s
$\mathcal{A}^+ =_{def} \mathcal{A} \mid \mathcal{A}^*$	a finite multiset of one or more $\mathcal{A}$'s
$A? =_{def} \mathbf{0} \vee \mathcal{A}$	optionally an $\mathcal{A}$
$\mathbf{T}$	anything

In general, we believe that a number of proposals for describing the shape of semistructured data can be embedded in our logic. Each such proposal usually comes with an efficient algorithm for checking membership or other properties. These efficient algorithms, of course, do not fall out automatically from a general framework.

6.2 Constraints

While types constrain the shape of data, it is often useful to constrain the values too; the canonical examples are key constraints and referential integrity constraints.

We have already provided an example of key constraint in TQL in Section 2.4, and we observed that many different notions of key have been studied for semistructured data. For example, Buneman et al. [9] define a notion of *relative* keys. Assume you have a set of books whose type, expressed as in the previous section, is:

$$\begin{array}{llll}
\textit{BOOKS} \vDash \textit{books} & [\ \textit{book} & [\ \textit{chapter}[\textit{number}[\mathbf{T}] \mid \textit{contents}[\mathbf{T}]\ ]* & \\
& &]* & \\
&] & &
\end{array}$$

we say that *number* is a key for *chapter* relative to *books.book*, and this means that, for each specific book, it is never the case that two different chapters have the same number. Of course, *number* is not an *absolute* key for *books.book.chapter*, since two different chapters (in two different books) may have the same number. This is expressed in TQL by the following formula.

$$BOOKS \vDash \neg books.book[\,.chapter.number[\mathcal{X}] \mid .chapter.number[\mathcal{X}]\,]$$

A positive version of the formula can be used to find any chapter number that violates the constraint, and the involved book $\mathcal{Y}$:

$$\begin{array}{ll} \textit{from} & BOOKS \vDash books.book(\mathcal{Y})[\,.chapter.number[\mathcal{X}] \mid .chapter.number[\mathcal{X}]\,] \\ \textit{select} & ReusedChapterNumbers[\,book[\mathcal{Y}] \mid number[\mathcal{X}]\,] \end{array}$$

The notion defined in [9] is slightly more complex. The relative key constraint we have shown is there described as (*books.book*(*chapter*,(*number*))), which is a special case of a more general constraint $(Q,(Q',(P_1,\ldots,P_n)))$.

$(Q,(Q',(P_1,\ldots,P_n)))$ specifies that, for each element e that can be reached through the path Q from the root (each book) and for each two different subelements e', e'', reachable from e through Q' (e.g., two chapters of the same book) one key-path P_i exists such that any subelement of e' reachable through P_i is different from any subelement of e'' reachable through P_i. This is quite long to express in first-order logic, and we even cheated a little, since the actual definition of [9] distinguishes node-equality, used to compare e' and e'', from value-equality, used to compare their P_i-reachable subelements.

In TQL logic, the same notion can be expressed, without cheating, as:

$$\forall \mathcal{X}_1 \ldots \forall \mathcal{X}_n.\ \neg.Q[\ .Q'[.P_1[\mathcal{X}_1] \wedge \ldots \wedge .P_n[\mathcal{X}_n]] \mid .Q'[.P_1[\mathcal{X}_1] \wedge \ldots \wedge .P_n[\mathcal{X}_n]]\ ]$$

Referential integrity constraints can be exemplified by considering the following schema, describing a list of books and a list of authors.

$$\begin{array}{l} books[\ book[\ author[auth\text{-}id[\mathbf{T}]]* \mid \mathbf{T}\ ]*\ ] \\ \mid authors[\ author[\ id[\mathbf{T}] \mid \mathbf{T}\ ]*\ ] \end{array}$$

Each author is identified by an *auth-id*; the referential constraint specifies that the *auth-id*'s have to be included into the actual *id*'s of registered authors. In TQL, this can be expressed as follows.

$$\forall \mathcal{X}.\ .books.book.author.auth\text{-}id[\mathcal{X}]\ \Rightarrow\ .authors.author.id[\mathcal{X}]$$

As a conclusion, TQL logic allows types and constraints to be easily specified, and, as we exemplified before, TQL allows one to write queries to check *whether* a constraint holds, to discover *where* a constraint *does not* hold, and also to discover *which* constraints hold (Section 2.4: which fields are keys?). The next step is *reasoning* about constraints (and types), for example using them to optimize queries and to pinpoint that some parts of a query are not compatible with some constraint, or that some constraints are not mutually compatible.

If we restrict ourselves to the TQL version of families of constraints that have already been studied, we can reuse known algorithms for deciding constraint implication; for example, we can rephrase the excellent study on the manipulation of key constraint of [10] in terms of the TQL logic. Of course, the real issue is the generalization of those result, to encopass a greater subset of TQL logic. We have preliminary results on this, and we also plan to exploit the emerging results about algorithms for checking the validity of ambient logic formulae ([12]).

7 Query Evaluation

In this section we define a query evaluation procedure. This procedure is really a refined semantics of queries, which is intermediate in abstraction between the semantics of Section 5.2 and an implementation algorithm, and may be seen as a specification of such implementation.

The core of query evaluation is the binder evaluation procedure, used to execute the *from* $Q \vDash \mathcal{A}$ part of a query. It takes the value I of Q and an input valuation ρ, and returns the set of all the valuations ρ' such that $I \in [\![\mathcal{A}]\!]_{\rho;\rho'}$, . The basic feature of this procedure is the fact that it does not compute 'one valuation at a time', in the style, for example, of a Prolog interpreter, but it is based on set manipulation: the set of all valuations associated with a pair I-$\mathcal{A}$ is obtained by combining, with set operations, the sets of valuations extracted by the immediate subformulae of $\mathcal{A}$. We chose set-based evaluation because it is the only approach that guarantees reasonable performace in presence of large amounts of data, hence is the forced choice for database-like applications. For this reason, our procedure is based on an algebra of tables (sets of valuations) and trees, and is a precise, although abstract, specification of the actual TQL implementation. The actual TQL implementation is described in [18, 19], and can be seen as the kernel of a realistic database-like implementation.

The procedure we describe here is abstract because it is based on the manipulation of sets of valuations which may be infinite. In the real implementation we adopt one specific finite representation of these infinite tables, in terms of a finite disjunction of a set of conjunctive constraints over the valuations, in the style of [27, 28]. We are not going to describe it here, but more information can be found in [18, 19]. Moreover, the real implementation directly implements the dualized logical operators indicated in the first table in Section 4.2. For the sake of simplicity, we assume here, on the contrary, that all derived operators are rewritten in terms of the basic operators (including, of course, negation) before execution.

Our query evaluation procedure shows how to directly evaluate a query to a resulting set of trees. In our actual implementation, instead, we translate the query into an expression over algebraic operators (which include also operators such as if-then-else, iteration and fixpoint). These expressions are first syntactically manipulated to enhance their performance; for example, negation is pushed down to the leaves of formulas, exploiting the dualized operators. Finally, they are evaluated. We ignore here issues of translation and manipulation of intermediate representations.

Any practical implementation of a query language is based on the use of efficiently implementable operators, such as relational join and union. We write our query evaluation procedure in this style as much as possible, at least for the basic operators that we consider here.

To describe the procedure, we first introduce an algebra over tables. Tables are sets of valuations (here called rows). We then use this algebra to define the evaluation procedure.

7.1 The Table Algebra

As in the previous sections, $\mathbf{V} = \mathcal{V}_1, ..., \mathcal{V}_n$ is a finite set of variables, and a row $\rho^{\mathbf{V}}$ with schema $\mathbf{V}$ is a function that maps each $\mathcal{V}_i$ to an element of $U(\mathcal{V}_i)$. A table with schema $\mathbf{V}$ is a set of rows over $\mathbf{V}$. We use $\mathbf{1}^{\mathbf{V}}$ to denote the largest table with schema $\mathbf{V}$, i.e. the set of all rows with schema $\mathbf{V}$, and $\mathcal{T}^{\mathbf{V}}$ for the set $\mathcal{P}(\mathbf{1}^{\mathbf{V}})$ of all the tables with schema $\mathbf{V}$. We use $R^{\mathbf{V}}$ as a meta-variable to range over $\mathcal{T}^{\mathbf{V}}$, i.e. $R^{\mathbf{V}} \in \mathcal{T}^{\mathbf{V}}$. We omit the superscript $\mathbf{V}$ when is irrelevant or is clear from the context.

When $\mathbf{V}$ is the empty set, we have only one row over $\mathbf{V}$, which we denote with ϵ; hence we have only two tables with schema $\emptyset$, the empty one, $\emptyset$, and the singleton, $\{\epsilon\} = \mathbf{1}^{\emptyset}$.

The table algebra is based on five primitive operators: union, complement, product, projection, and restriction, each carrying schema information. They correspond to the standard operations of relational algebra.

The operators of table algebra:

	$R^{\mathbf{V}} \cup^{\mathbf{V}} R'^{\mathbf{V}}$	$=_{def}$	$R^{\mathbf{V}} \cup R'^{\mathbf{V}}$	$\subseteq \mathbf{1}^{\mathbf{V}}$
	$Co^{\mathbf{V}}(R^{\mathbf{V}})$	$=_{def}$	$\mathbf{1}^{\mathbf{V}} \setminus R^{\mathbf{V}}$	$\subseteq \mathbf{1}^{\mathbf{V}}$
$\mathbf{V}' \cap \mathbf{V} = \emptyset$:	$R^{\mathbf{V}} \times^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$	$=_{def}$	$\{\rho;\rho' \mid \rho \in R^{\mathbf{V}},\ \rho' \in R'^{\mathbf{V}'}\}$	$\subseteq \mathbf{1}^{\mathbf{V}\cup\mathbf{V}'}$
$\mathbf{V}' \subseteq \mathbf{V}$:	$\prod^{\mathbf{V}}_{\mathbf{V}'} R^{\mathbf{V}}$	$=_{def}$	$\{\rho' \mid \rho' \in \mathbf{1}^{\mathbf{V}'},\ \exists \rho \in R^{\mathbf{V}}.\ \rho \supseteq \rho'\}$	$\subseteq \mathbf{1}^{\mathbf{V}'}$
$FV(\eta,\eta') \subseteq \mathbf{V}$:	$\sigma^{\mathbf{V}}_{\eta\sim\eta'} R^{\mathbf{V}}$	$=_{def}$	$\{\rho \mid \rho \in R^{\mathbf{V}},\ \rho_+(\eta) \sim \rho_+(\eta')\}$	$\subseteq \mathbf{1}^{\mathbf{V}}$

The table union $R^{\mathbf{V}} \cup^{\mathbf{V}} R'^{\mathbf{V}}$ is defined as the set-theoretic union of two tables with the same schema $\mathbf{V}$.

The table complement $Co^{\mathbf{V}}(R^{\mathbf{V}})$ is defined as the set-theoretic difference $\mathbf{1}^{\mathbf{V}} \setminus R^{\mathbf{V}}$.

If $R^{\mathbf{V}}$ and $R'^{\mathbf{V}'}$ are two tables whose schemas are disjoint, their table cartesian product $R^{\mathbf{V}} \times^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$ is defined as the set containing all rows obtained by concatenating each row of $R^{\mathbf{V}}$ with each row of $R'^{\mathbf{V}'}$. The result has schema $\mathbf{V} \cup \mathbf{V}'$.

If $\mathbf{V}'$ is a subset of $\mathbf{V}$, the projection $\prod^{\mathbf{V}}_{\mathbf{V}'} R^{\mathbf{V}}$ is defined as the set of all rows in $R^{\mathbf{V}}$ restricted to the variables in $\mathbf{V}'$.

Let $\rho^{\mathbf{V}}_+$ be the function that coincides with $\rho^{\mathbf{V}}$ over $\mathbf{V}$, and maps every $\eta \notin \mathbf{V}$ to η. If $FV(\eta,\eta') \subseteq \mathbf{V}$, then the restriction $\sigma^{\mathbf{V}}_{\eta\sim\eta'} R^{\mathbf{V}}$ is the set

$$\{\rho^{\mathbf{V}} \mid \rho^{\mathbf{V}} \in R^{\mathbf{V}} \ \mathit{and}\ \rho^{\mathbf{V}}_+(\eta) \sim \rho^{\mathbf{V}}_+(\eta')\}\ ,$$

where $\sim$ is a label comparison operator, as in Section 4. When both η and η' are two constants n, n', then $\sigma^{\mathbf{V}}_{\eta\sim\eta'} R^{\mathbf{V}}$ is either $R^{\mathbf{V}}$, if $n \sim n'$, or $\emptyset$, if $\neg(n \sim n')$.

We will also use some derived operators, defined in the following table.

Table algebra, derived operators:

$\mathbf{V} \subseteq \mathbf{V}'$:	$Ext^{\mathbf{V}}_{\mathbf{V}'}(R^{\mathbf{V}})$	$=_{def}$	$R^{\mathbf{V}} \times^{\mathbf{V},\mathbf{V}'\setminus\mathbf{V}} \mathbf{1}^{\mathbf{V}'\setminus\mathbf{V}}$	$\subseteq \mathbf{1}^{\mathbf{V}'}$
	$R^{\mathbf{V}} \cap^{\mathbf{V}} R'^{\mathbf{V}}$	$=_{def}$	$Co^{\mathbf{V}}(Co^{\mathbf{V}}(R^{\mathbf{V}}) \cup^{\mathbf{V}} Co^{\mathbf{V}}(R'^{\mathbf{V}}))$	$\subseteq \mathbf{1}^{\mathbf{V}}$
	$R^{\mathbf{V}} \bowtie^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$	$=_{def}$	$Ext^{\mathbf{V}}_{\mathbf{V}\cup\mathbf{V}'}(R^{\mathbf{V}}) \cap^{\mathbf{V}\cup\mathbf{V}'} Ext^{\mathbf{V}'}_{\mathbf{V}\cup\mathbf{V}'}(R'^{\mathbf{V}'})$	$\subseteq \mathbf{1}^{\mathbf{V}\cup\mathbf{V}'}$

The operator $R^{\mathbf{V}} \bowtie^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$ is well-known in the database field. It is called 'natural join', and can be also defined as follows: a row $\rho^{\mathbf{V}\cup\mathbf{V}'}$ belongs to $R^{\mathbf{V}} \bowtie^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$ iff its restriction to $\mathbf{V}$ is in $R^{\mathbf{V}}$ and its restriction to $\mathbf{V}'$ is in $R'^{\mathbf{V}'}$. One important property of natural join is that it always yields finite tables when is applied to finite tables, even if its definition uses the extension operator. Moreover, the optimization of join has been extensively studied; for this reason we will use this operator, rather than extension plus intersection, in the definition of our query evaluation procedure.

7.2 Query Evaluation

We specify here an evaluation procedure $\mathcal{Q}(Q)_\rho$ that, given a query Q and a row ρ that specifies a value for each free variable of Q, evaluates the corresponding information tree. A closed query "*from* $Q \vDash \mathcal{A}$ *select* Q'" is evaluated by first evaluating Q to an information tree I. The pair $I, \mathcal{A}$ is then evaluated to yield a table $R^{\mathbf{V}}$ whose schema contains all the free variables in $\mathcal{A}$. Finally, Q' is evaluated once for each row ρ of $R^{\mathbf{V}}$; all the resulting information trees are combined using $|$, to obtain the query result. This process is expressed in the last case of the table below.

The first part of the table describes how a quadruple $I, \mathcal{A}, \rho^{\mathbf{V}}, \gamma$ is evaluated by a binder evaluation procedure $\mathcal{B}$ to return a table with schema $\mathcal{S}(\mathcal{A}, \mathbf{V}, \hat{\gamma})$. The schema function $\mathcal{S}$ is specified in the table that follows, and enjoys the property that $\mathcal{S}(\mathcal{A}, \mathbf{V}, \hat{\epsilon}) = FV(\mathcal{A}) \setminus \mathbf{V}$. Here γ is an environment that maps recursion variables ξ to functions from information trees to tables, and, for any γ, $\hat{\gamma}$ specifies its type, as follows:

$$\forall \xi \in dom(\gamma).\ \gamma(\xi) : \mathcal{IT} \to \mathcal{T}^{\hat{\gamma}(\xi)}.$$

The notation $\{(x \mapsto n)\}$ represents a table that contains only the row $(x \mapsto n)$, and similarly for $\{(\mathcal{X} \mapsto I)\}$.

Binder and query evaluation

$$
\begin{array}{lcll}
\mathcal{B}(I, \mathbf{0})_{\rho^{\mathbf{V}},\gamma} & = & \mathit{if}\ I = \mathbf{0}\ \mathit{then}\ \{\epsilon\}\ \mathit{else}\ \emptyset & \\
\mathcal{B}(I, n[\mathcal{A}])_{\rho^{\mathbf{V}},\gamma} & = & \mathit{if}\ I = n[I']\ \mathit{then}\ \mathcal{B}(I', \mathcal{A})_{\rho^{\mathbf{V}},\gamma}\ \mathit{else}\ \emptyset & \\
\mathcal{B}(I, x[\mathcal{A}])_{\rho^{\mathbf{V}},\gamma} & = & \mathcal{B}(I, \rho^{\mathbf{V}}(x)[\mathcal{A}])_{\rho^{\mathbf{V}},\gamma} & \text{if } x \in \mathbf{V} \\
\mathcal{B}(I, x[\mathcal{A}])_{\rho^{\mathbf{V}},\gamma} & = & \mathit{if}\ I = n[I']\ \mathit{then}\ \{(x \mapsto n)\} \bowtie^{\{x\}, \mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})} \mathcal{B}(I', \mathcal{A})_{\rho^{\mathbf{V}},\gamma} \mathit{else}\ \emptyset & \text{if } x \notin \mathbf{V} \\
\mathcal{B}(I, \mathcal{A} \mid \mathcal{B})_{\rho^{\mathbf{V}},\gamma} & = & \bigcup^{\mathcal{S}(\mathcal{A}|\mathcal{B},\mathbf{V},\hat{\gamma})}_{I',I'' \in \{I',I'' \mid I'|I''=I\}} (\mathcal{B}(I', \mathcal{A})_{\rho^{\mathbf{V}},\gamma} \bowtie^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma}), \mathcal{S}(\mathcal{B},\mathbf{V},\hat{\gamma})} \mathcal{B}(I'', \mathcal{B})_{\rho^{\mathbf{V}},\gamma}) & \\
\mathcal{B}(I, \mathbf{T})_{\rho^{\mathbf{V}},\gamma} & = & \{\epsilon\} & \\
\mathcal{B}(I, \neg\mathcal{A})_{\rho^{\mathbf{V}},\gamma} & = & Co^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})}(\mathcal{B}(I, \mathcal{A})_{\rho^{\mathbf{V}},\gamma}) & \\
\mathcal{B}(I, \mathcal{A} \wedge \mathcal{B})_{\rho^{\mathbf{V}},\gamma} & = & \mathcal{B}(I, \mathcal{A})_{\rho^{\mathbf{V}},\gamma} \bowtie^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma}), \mathcal{S}(\mathcal{B},\mathbf{V},\hat{\gamma})} \mathcal{B}(I, \mathcal{B})_{\rho^{\mathbf{V}},\gamma} & \\
\mathcal{B}(I, \mathcal{X})_{\rho^{\mathbf{V}},\gamma} & = & \mathit{if}\ I = \rho^{\mathbf{V}}(\mathcal{X})\ \mathit{then}\ \{\epsilon\}\ \mathit{else}\ \emptyset & \text{if } \mathcal{X} \in \mathbf{V} \\
\mathcal{B}(I, \mathcal{X})_{\rho^{\mathbf{V}},\gamma} & = & \{(\mathcal{X} \mapsto I)\} & \text{if } \mathcal{X} \notin \mathbf{V} \\
\mathcal{B}(I, \exists \mathcal{X}.\ \mathcal{A})_{\rho^{\mathbf{V}},\gamma} & = & \prod^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})}_{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma}) \setminus \{\mathcal{X}\}} \mathcal{B}(I, \mathcal{A})_{\rho^{\mathbf{V}},\gamma} & \\
\mathcal{B}(I, \exists x.\ \mathcal{A})_{\rho^{\mathbf{V}},\gamma} & = & \prod^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})}_{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma}) \setminus \{x\}} \mathcal{B}(I, \mathcal{A})_{\rho^{\mathbf{V}},\gamma} & \\
\mathcal{B}(I, \eta \sim \eta')_{\rho^{\mathbf{V}},\gamma} & = & \sigma^{\mathcal{S}(\eta \sim \eta',\mathbf{V},\hat{\gamma})}_{\rho_+(\eta) \sim \rho_+(\eta')} \mathbf{1}^{\mathcal{S}(\eta \sim \eta',\mathbf{V},\hat{\gamma})} & \\
\mathcal{B}(I, \mu\xi.\mathcal{A})_{\rho^{\mathbf{V}},\gamma} & = & Fix(\lambda M \in \mathcal{IT} \to \mathcal{T}^{\mathcal{S}(\mu\xi.\mathcal{A},\mathbf{V},\hat{\gamma})}.\lambda \mathcal{Y}.\mathcal{B}(\mathcal{Y}, \mathcal{A})_{\rho^{\mathbf{V}},\gamma[\xi \mapsto M]})(I) & \\
\mathcal{B}(I, \xi)_{\rho^{\mathbf{V}},\gamma} & = & \gamma(\xi)(I) &
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{Q}(\mathcal{X})_{\rho^{\mathbf{V}}} & = \rho^{\mathbf{V}}(\mathcal{X}) \\
\mathcal{Q}(\mathbf{0})_{\rho^{\mathbf{V}}} & = \mathbf{0} \\
\mathcal{Q}(Q \mid Q')_{\rho^{\mathbf{V}}} & = \mathcal{Q}(Q)_{\rho^{\mathbf{V}}} \mid \mathcal{Q}(Q')_{\rho^{\mathbf{V}}} \\
\mathcal{Q}(m[Q])_{\rho^{\mathbf{V}}} & = m[\mathcal{Q}(Q)_{\rho^{\mathbf{V}}}] \\
\mathcal{Q}(x[Q])_{\rho^{\mathbf{V}}} & = \rho^{\mathbf{V}}(x)[\mathcal{Q}(Q)_{\rho^{\mathbf{V}}}] \\
\mathcal{Q}(f(Q))_{\rho^{\mathbf{V}}} & = f(\mathcal{Q}(Q)_{\rho^{\mathbf{V}}}) \\
\mathcal{Q}(\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q')_{\rho^{\mathbf{V}}} & = \text{let } I = \mathcal{Q}(Q)_{\rho^{\mathbf{V}}} \text{ and } R^{FV(\mathcal{A}) \setminus \mathbf{V}} = \mathcal{B}(I, \mathcal{A})_{\rho^{\mathbf{V}},\epsilon} \\
 & \quad \text{in } Par_{\rho' \in R^{FV(\mathcal{A}) \setminus \mathbf{V}}}\ \mathcal{Q}(Q')_{(\rho^{\mathbf{V}};\rho')}
\end{array}
$$

The schema function $\mathcal{S}$

$$
\begin{array}{lcl}
\mathcal{S}(\mathbf{0},\mathbf{V},\Gamma) & = & \emptyset \\
\mathcal{S}(n[\mathcal{A}],\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \\
\mathcal{S}(x[\mathcal{A}],\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \cup (\{x\} \setminus \mathbf{V}) \\
\mathcal{S}(\mathcal{A} \mid \mathcal{B},\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \cup \mathcal{S}(\mathcal{B},\mathbf{V},\Gamma) \\
\mathcal{S}(\mathbf{T},\mathbf{V},\Gamma) & = & \emptyset \\
\mathcal{S}(\neg\mathcal{A},\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \\
\mathcal{S}(\mathcal{A} \wedge \mathcal{B},\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \cup \mathcal{S}(\mathcal{B},\mathbf{V},\Gamma) \\
\mathcal{S}(\mathcal{X},\mathbf{V},\Gamma) & = & \{\mathcal{X}\} \setminus \mathbf{V} \\
\mathcal{S}(\exists \mathcal{X}.\ \mathcal{A},\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \setminus \{\mathcal{X}\} \\
\mathcal{S}(\exists x.\ \mathcal{A},\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma) \setminus \{x\} \\
\mathcal{S}(\eta \sim \eta',\mathbf{V},\Gamma) & = & FV(\eta,\eta') \setminus \mathbf{V} \\
\mathcal{S}(\mu\xi.\mathcal{A},\mathbf{V},\Gamma) & = & \mathcal{S}(\mathcal{A},\mathbf{V},\Gamma[\xi \mapsto \emptyset]) \\
\mathcal{S}(\xi,\mathbf{V},\Gamma) & = & \Gamma(\xi)
\end{array}
$$

We explain now the evaluation procedure. In this page, the sentence 'ρ satisfies $I \vDash \mathcal{A}$' means that $I \in [\![\mathcal{A}]\!]_{\rho,\delta}$, for some δ (we will ignore δ in the following informal discussion).

The $\mathbf{0}$ formula is evaluated by testing whether the subject is the empty tree, and returning either the trivial singleton $\mathbf{1}^{\emptyset}$ or the empy set. Hence, in a query *from* $Q \vDash \mathbf{0}$ *select isZero*, the *select isZero* branch will be executed once if, and only if, Q evaluates to $\mathbf{0}$. More generally, if $\rho^{\mathbf{V}}$ is the input valuation and $\mathbf{V}' = FV(\mathcal{A}) \setminus \mathbf{V}$, then $\mathbf{1}^{\mathbf{V}'}$ is the set of valuations that corresponds to *truth*, i.e. to the case when $[\![Q]\!]_{\rho^{\mathbf{V}}} \in [\![\mathcal{A}]\!]_{\rho^{\mathbf{V}};\rho'^{\mathbf{V}'},\epsilon}$ holds for any $\rho'^{\mathbf{V}'}$. Similarly, $\emptyset$ is the table that corresponds to to the case when $[\![Q]\!]_{\rho^{\mathbf{V}}} \in [\![\mathcal{A}]\!]_{\rho^{\mathbf{V}};\rho'^{\mathbf{V}'},\epsilon}$ does *not* hold, for any $\rho'^{\mathbf{V}'}$.

The formula $n[\mathcal{A}]$ tests wether I is an edge with label n. If it is not, the empty set of binders is returned. Otherwise, the contents of I are matched against $\mathcal{A}$. A formula $x[\mathcal{A}]$ is evaluated in the same way if x is already bound by the input valuation. Otherwise, if I is not an edge, no binder is returned. If $I = n[I']$, then the result is built by joining $\{(x \mapsto n)\}$ with the result $R'^{\mathbf{V}'}$ of matching I' with $\mathcal{A}$. By definition of natural join, if x is not bound by $R'^{\mathbf{V}'}$, i.e. $x \notin \mathbf{V}'$, then $\{(x \mapsto n)\} \bowtie R'^{\mathbf{V}'}$ is just a cartesian product; otherwise, it is equivalent to $\sigma_{x \sim n} R'^{\mathbf{V}'}$.

Truth and negation need no explanation. Set complement of a finite table produces an infinite table, and is, in general, quite expensive to compute. For this reason, a realistic implementation would try and minimize the use of this operator.

Conjunction corresponds to natural join: a valuation satisfies $I \vDash \mathcal{A} \wedge \mathcal{B}$ if, and only if, its restriction to the free variables of $\mathcal{A}$ satisfies $I \vDash \mathcal{A}$ and its restriction to the free variables of $\mathcal{B}$ satisfies $I \vDash \mathcal{B}$.

A valuation satisfies $I \vDash \mathcal{A} \mid \mathcal{B}$ if there exists a decomposition $I' \mid I''$ of I such that $I' \vDash \mathcal{A}$ and $I'' \vDash \mathcal{A}$. For this reason, we try all possible decompositions of the subject I, and, for each of them, we compute the natural join of the sets of valuations for $I' \vDash \mathcal{A}$ and $I'' \vDash \mathcal{A}$. Any time a decomposition contributes some valuations, we put them in the result (this is the aim of the big union outside). Since an information tree with n top-level branches admits 2^n different decompositions, this operation is horribly expensive. However, if $\mathcal{A} \Rightarrow \exists x.\ x[\mathbf{T}]$, then $\mathcal{A}$ can only be satisfied by a one-edge subtree, hence only the n different decompositions with shape $n[I'] \mid I''$ have to be tried. The actual implementation systematically exploits this observation; as a result, every decomposition that appears in the queries presented in Section 2 is actually executed in linear time. This optimization is based on a quite simple algorithm which tries and verify whether $\mathcal{A} \Rightarrow \exists x.\ x[\mathbf{T}]$; the simplicity and effectiveness of this algorithm is clearly a consequence of the fact that all the operators that appear in a TQL binder have a simple logic-based interpretation.

If $\mathcal{X}$ is bound by the input valuation, then $Q \vDash \mathcal{X}$ checks whether $[\![Q]\!]_\rho = \rho(\mathcal{X})$. If $\mathcal{X}$ is not bound, the valuation $(\mathcal{X} \mapsto [\![Q]\!]_{\rho^{\mathbf{V}}})$ is returned.

Projection is used to evaluate existential quantification, since, by definition:

$$\begin{array}{lcl} \rho \in \prod_{\mathbf{V}\setminus\{x\}}^{\mathbf{V}} R^{\mathbf{V}} & \Leftrightarrow & \exists n \in \Lambda.\ (\rho; (x \mapsto n)) \in R^{\mathbf{V}} \\ \rho \in \prod_{\mathbf{V}\setminus\{\mathcal{X}\}}^{\mathbf{V}} R^{\mathbf{V}} & \Leftrightarrow & \exists I \in \mathcal{IT}.\ (\rho; (\mathcal{X} \mapsto I)) \in R^{\mathbf{V}} \end{array}$$

Since the rule for comparisons $\eta \sim \eta'$ is subtle, we expand here some special cases. In general, it returns a table whose schema corresponds to $FV(\{\eta, \eta'\} \setminus \mathbf{V})$. Hence, if both η and η' are either constant or bound by $\rho^{\mathbf{V}}$, it returns a table with an empty schema, i.e. either $\mathbf{1}^{\emptyset}$ or $\emptyset$ (cases 3 and 5). If both η and η' are unbound variables, it returns the infinite table that defines the $\sim$ operator; for example, when $\sim$ is equality, it returns the diagonal table that maps η, η' to n, n (case 1).

Some special cases of comparison evaluation

1. $\mathcal{B}(I, x \sim x')_{\rho^{\mathbf{V}},\gamma} = \sigma^{\{x,x'\}}_{x\sim x'} \mathbf{1}^{\{x,x'\}}$ if $x \notin \mathbf{V}, x' \notin \mathbf{V}$
2. $\mathcal{B}(I, x \sim x')_{\rho^{\mathbf{V}},\gamma} = \sigma^{\{x\}}_{x\sim\rho^{\mathbf{V}}(x')} \mathbf{1}^{\{x\}}$ if $x \notin \mathbf{V}, x' \in \mathbf{V}$
3. $\mathcal{B}(I, x \sim x')_{\rho^{\mathbf{V}},\gamma} = \sigma^{\emptyset}_{\rho^{\mathbf{V}}(x)\sim\rho^{\mathbf{V}}(x')} \mathbf{1}^{\emptyset}$ if $x \in \mathbf{V}, x' \in \mathbf{V}$
4. $\mathcal{B}(I, x \sim n)_{\rho^{\mathbf{V}},\gamma} = \sigma^{\{x\}}_{x\sim n} \mathbf{1}^{\{x\}}$ if $x \notin \mathbf{V}$
5. $\mathcal{B}(I, n \sim n')_{\rho^{\mathbf{V}},\gamma} = \sigma^{\emptyset}_{n\sim n'} \mathbf{1}^{\emptyset}$ (*i.e. if* $n \sim n'$ *then* $\{\epsilon\}$ *else* $\emptyset$)

Finally, we have the recursive case. $Fix(\lambda M.\lambda\mathcal{Y}.R^{\mathbf{V}})$ denotes the minimal fixpoint of a function mapping M to $\lambda\mathcal{Y}.R^{\mathbf{V}}$; in a programming language, you would express this as:

$$\textit{letrec } M = \textit{fun}(\mathcal{Y}).R^{\mathbf{V}} \textit{ in } M$$

So, we first transform the recursive formula into a recursive function from information trees to tables, and then we apply this recursive function to the subject I. The correctness of this evaluation technique is far from obvious, and is proven in detail in the Appendix.

To prove the correctness of the evaluation procedure we first need a couple of lemmas to state that types are corrects.

Lemma 1

$$\mathcal{S}(\mathcal{A}, \mathbf{V}, \epsilon) = FV(\mathcal{A}) \setminus \mathbf{V}$$

Proof By induction and case analysis. □

Lemma 2

$$\mathcal{S}(\mu\xi.\mathcal{A}, \mathbf{V}, \Gamma) = \mathcal{S}(\mathcal{A}, \mathbf{V}, \Gamma[\xi \mapsto \mathcal{S}(\mu\xi.\mathcal{A}, \mathbf{V}, \Gamma)])$$

Proof By induction on $\mathcal{A}$. When $\mathcal{A} = \mu\zeta.\mathcal{B}$ we commute the binding for ξ and ζ in $\Gamma[\xi \mapsto \ldots][\zeta \mapsto \emptyset]$. When $\mathcal{A} = \xi$ we have:

$$\mathcal{S}(\xi, \mathbf{V}, \Gamma[\xi \mapsto \mathcal{S}(\mu\xi.\xi, \mathbf{V}, \Gamma)]) \ = \text{(by def.)} \ \ (\Gamma[\xi \mapsto \mathcal{S}(\mu\xi.\xi, \mathbf{V}, \Gamma)])(\xi) \ = \ \mathcal{S}(\mu\xi.\xi, \mathbf{V}, \Gamma)$$

All other cases are immediate by induction. □

Lemma 3

$$\mathcal{B}(I,\mathcal{A})_{\rho^{\mathbf{V}},\gamma} \in \mathcal{T}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})}.$$

Proof By induction on $\mathcal{A}$ and by cases. The only non-trivial case is recursion, where we resort to Lemma 2. $\square$

We can now state the main lemma, that specifies the correcteness of the binder evaluation procedure.

Lemma 4 *Let $\mathcal{A}$ be a formula, $\mathbf{V}$ be a set of variables, let Ξ be a set $\{\xi_i\}^{i\in I}$ of recursion variables that includes those that are free in $\mathcal{A}$, and let γ be a function defined over Ξ such that, for every ξ_i, $\gamma(\xi_i)\in\mathcal{IT}\rightarrow\mathcal{T}^{\hat{\gamma}(\xi_i)}$, where $\hat{\gamma}(\xi_i)$ is disjoint from $\mathbf{V}$. then:*

$$\forall\rho\in\mathbf{1}^{\mathbf{V}}, I\in\mathcal{IT}.\quad \mathcal{B}(I,\mathcal{A})_{\rho,\gamma} \;=\; \{\rho' \mid \rho'\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})},\ I\in[\![\mathcal{A}]\!]_{(\rho;\rho'),\,d(\gamma,\rho')}\}$$

where $d(\gamma,\rho)=\lambda\xi\!:\!\Xi.\{I \mid \rho\in\gamma(\xi)(I)\}$.

Proof Not trivial; see the Appendix. $\square$

Finally, the following theorem states that the query evaluation procedure is equivalent to the query semantics of Section 5.2.

Theorem 1 $\forall Q,\ \mathbf{V}\supseteq FV(Q),\ \rho^{\mathbf{V}}.\ \ \mathcal{Q}(Q)_{\rho^{\mathbf{V}}}=[\![Q]\!]_{\rho^{\mathbf{V}}}$

Proof By induction on Q and by cases.

Assume $Q = \textit{from } Q'\vDash\mathcal{A}\textit{ select } Q''$ and consider the definition of $\mathcal{Q}(\textit{from } Q'\vDash\mathcal{A}\textit{ select } Q'')_{\rho^{\mathbf{V}}}$.

$\mathcal{Q}(\textit{from } Q'\vDash\mathcal{A}\textit{ select } Q'')_{\rho^{\mathbf{V}}}$
$\quad = \text{let } I=\mathcal{Q}(Q')_{\rho^{\mathbf{V}}} \text{ and } R^{FV(\mathcal{A})\setminus\mathbf{V}}=\mathcal{B}(I,\mathcal{A})_{\rho^{\mathbf{V}},\epsilon} \text{ in } Par_{\rho'\in R^{FV(\mathcal{A})\setminus\mathbf{V}}}\ \mathcal{Q}(Q'')_{(\rho^{\mathbf{V}};\rho')}$

By induction, $\mathcal{Q}(Q')_{\rho^{\mathbf{V}}}=[\![Q']\!]_{\rho^{\mathbf{V}}}$. Hence, by Lemma 4, and using $\Xi=\emptyset$ ($\mathcal{A}$ contains no free recursion variables) we have:

$$R^{FV(\mathcal{A})\setminus\mathbf{V}}=\mathcal{B}([\![Q']\!]_{\rho^{\mathbf{V}}},\mathcal{A})_{\rho^{\mathbf{V}},\epsilon}=\{\rho' \mid \rho'\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\epsilon})},\ [\![Q']\!]_{\rho^{\mathbf{V}}}\in[\![\mathcal{A}]\!]_{(\rho^{\mathbf{V}};\rho'),\,\epsilon}\} \qquad \text{(a)}$$

By induction, we also have that, for any ρ, $\mathcal{Q}(Q'')_\rho=[\![Q'']\!]_\rho$. Hence:

$$\begin{array}{ll}
 & \mathcal{Q}(\textit{from } Q'\vDash\mathcal{A}\textit{ select } Q'')_{\rho^{\mathbf{V}}}\\
\text{by def.} & = Par_{\rho''\in R^{FV(\mathcal{A})\setminus\mathbf{V}}}\ \mathcal{Q}(Q'')_{(\rho^{\mathbf{V}};\rho'')}\\
\text{by (a)} & = Par_{\rho''\in\{\rho'' \mid \rho''\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\epsilon})},\ [\![Q']\!]_{\rho^{\mathbf{V}}}\in[\![\mathcal{A}]\!]_{(\rho^{\mathbf{V}};\rho''),\,\epsilon}\}}\ [\![Q'']\!]_{(\rho^{\mathbf{V}};\rho'')}\\
\text{let } \rho'^{\mathbf{V}'}=\rho^{\mathbf{V}};\rho''\text{:} & = Par_{\rho'^{\mathbf{V}'}\in\{\rho^{\mathbf{V}};\rho'' \mid \rho''\in\mathbf{1}^{FV(\mathcal{A})\setminus\mathbf{V}},\ [\![Q']\!]_{\rho^{\mathbf{V}}}\in[\![\mathcal{A}]\!]_{(\rho^{\mathbf{V}};\rho''),\,\epsilon}\}}\ [\![Q'']\!]_{(\rho'^{\mathbf{V}'})}\\
\text{by } \rho'^{\mathbf{V}'}=\rho^{\mathbf{V}};\rho''\text{:} & = Par_{\rho'^{\mathbf{V}'}\in\{\rho'^{\mathbf{V}'} \mid \mathbf{V}'=\mathbf{V}\cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'}\supseteq\rho^{\mathbf{V}},\ [\![Q']\!]_{\rho^{\mathbf{V}}}\in[\![\mathcal{A}]\!]_{(\rho'^{\mathbf{V}'}),\,\epsilon}\}}\ [\![Q'']\!]_{(\rho'^{\mathbf{V}'})}\\
\text{by def.} & = [\![\textit{from } Q'\vDash\mathcal{A}\textit{ select } Q'']\!]_{\rho^{\mathbf{V}}}
\end{array}$$

The other cases are immediate.

$\square$

8 Comparisons with Related Proposals

In this paper we describe a logic, a query language, and an abstract evaluation mechanism.

The tree logic can be compared with standard first order formalizations of labelled trees. Using the terminology of [4], we can encode a labeled tree with a relation *Ref(source:OID, label:*Λ*, destination:OID)*. The nodes of the tree are the OIDs (Object IDentifiers) that appear in the *source* and *destination* columns, and any tuple in the relation represents an edge, with label *label*. Of course, such a relation can represent a graph as well as a tree. It represents a forest if *destination* is a key for the relation, and if there exists an order relation on the OIDs such that, in any tuple, the *source* strictly precedes the *destination*.

First order formulas defined over this relation already constitute a logical language to describe tree properties. Trees are represented here by the OID of their root. We can say that, for example, "the tree x is $t[]$" by saying:

$$\exists y.\ \mathit{Ref}(x,t,y) \wedge (\forall y', y''.\ \neg \mathit{Ref}(y,y',y'')) \wedge (\forall x', x''.\ x'' \neq y \Rightarrow \neg \mathit{Ref}(x,x',x''))$$

There are some differences with our approach. First, our logic is 'modal', a term which we use to say that a formula $\mathcal{A}$ is always about one specific 'subject', that is the part of the database currently being matched against $\mathcal{A}$. First order logic, instead, does not have an implicit subject: one can, and must, name a subject. For example, our modal formula $t[]$ implicitly describes the 'current tree', while its translation into first order logic, given above, gives a name x to the tree it describes.

Being 'modal' is neither a merit nor a fault, in itself; it is merely a difference. Modality makes it easier to decribe just one tree and its structure, whereas it makes it more difficult to describe a relationship between two different trees.

Apart from modality, another feature of the ambient logic is that its fundamental operators deal with one-step paths ($t[\mathcal{A}]$) and with the composition of trees ($\mathcal{A} \mid \mathcal{A}'$), whereas the first order approach describes everything in terms of one-step paths ($\mathit{Ref}(o1,t,o2)$). Composition is a powerful operator, at least for the following purposes:

- it makes it easy to say that two different subtrees both satisfy a property, without using node or edge identity, as in the following sentence, that specifies that *title* is not a key: $\exists \mathcal{X}.\ .\mathit{title}[\mathcal{X}] \mid .\mathit{title}[\mathcal{X}]$;
- it makes it easy to describe record-like structures both partially ($b[] \mid c[] \mid \mathbf{T}$ means: contains $b[]$, $c[]$, and possibly more fields) and completely ($b[] \mid c[]$ means: contains $b[], c[]$ and only $b[], c[]$); complete descriptions are difficult in the path based approach;
- it makes it possible to bind a variable to 'the rest of the record', as in 'X is everything but the title': $\mathit{paper}[\mathit{title}[\mathbf{T}] \mid X]$.

The query language we described derives its essential *from-select* structure from set-theoretics comprehension, in the SQL tradition, and this makes it similar to other query languages for semistructured data, such as StruQL [21, 22], Lorel [6, 25], XML-QL [20], Quilt [16], XQuery [7] and, to some extent, YATL [17]. An in-depth comparison between the XML-QL, YATL, and Lorel languages is carried out in [23], based on the analysis of thirteen typical queries. In [24] we wrote down those same queries in TQL; the result of this comparison is that, for the thirteen queries in [23], their TQL expression is quite similar to the corresponding XML-QL, with a couple of exceptions. First, those XML-QL queries that, in [23], are expressed using Skolem functions, have to be expressed in a different way in TQL, since we do not have Skolem functions in the current version of TQL. However, our Skolem-free version of these queries is not complex. Second, XML-QL does not seem to have a general way of expressing universal quantification, and this problem shows up in the query that asks for pairs of books with the same set of authors; this is rather complex to express in XML-QL, but it is not difficult in TQL. Another related class of queries that are simpler to express using TQL are those related to the non-existence of paths, such as 'find all the papers with no title' or 'find all the papers whose only author, if any, is Ghelli'. Lorel does not have these problems, since it allows universal quantification. Quilt, XQuery, and XDuce [26] are Turing complete, hence are more expressive than the other languages we cited here. However, the binding mechanism of Quilt and XQuery share the limitations of the path-based approaches of the other languages of the StruQL-Lorel-UnQL family.

One important feature of TQL is that it has a clean semantic interpretation, which pays off in several ways. First, the semantics makes it possible to prove the correctness and completeness of a specific implementation. Moreover, it simplifies the task of proving equivalences between different logic formulas or queries. To our knowledge, no such formal semantics has been defined for YATL. The semantics of Lorel has been defined, but looks quite involved, because of their extensive use of coercions. The semantics of XQuery has been defined too, but is intermediate, in spirit, between our notions of semantics (Section 5.2) and of abstract implementation specification (Section 7).

9 Conclusions and Future Directions

We defined a query language based on a logic that can also be used to describe types and constraints. This paves the way for optimization and error checking procedures based upon the search of parts of the query binder that are implied by the known constraints, or are not compatible with them. To this aim we are studying decidable subsets of the ambient logic, and also non-complete procedures for deciding the validity of non-decidable subsets of the logic.

Our query language operates on information represented as unordered trees. One can take different views of how information should be represented. For example as ordered trees, as in XML, or as unordered graphs, as in semistructured data. We believe that each choice of representation would lead to a (slightly different) logic and a query language along the lines described here. We are currently looking at some of these options.

There are currently many proposals for regular pattern languages for semistructured data, many having in common the desire to describe tree shapes and not just linear paths. Given the expressive power of general recursive formulas $\mu\xi.\mathcal{A}$, we believe we can capture many such proposals, even though an important part of those proposals is to describe efficient matching techniques.

In this study we have exploited a subset of the ambient logic. The ambient logic, and the calculus, also offer operators to specify and perform tree updates [13]. Possible connections with semistructured data updates should be explored.

An implementation of TQL, based on the implementation model we described, is available from `http://tql.di.unipi.it/tql`. The current prototype can be used to query XML documents accessible through files or through web servers.

Acknowledgements Andrew D. Gordon contributed to this work with many useful suggestions. Giorgio Ghelli was partially supported by Microsoft Research, and by the E.U. workgroup APPSEM.

References

[1] XML Query use cases. http://www.w3.org/TR/xmlquery-use-cases, 2002.

[2] XML Schema. http://www.w3.org/XML/Schema, 2002.

[3] XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/TR/query-datamodel, 2002.

[4] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the WEB: From Relations to Semistructured Data and XML.* Morgan Kaufmann, San Mateo, CA, October 1999.

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

[6] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[7] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jrme Simon, and Mugur Stefanescu. XQuery 1.0: An XML query language. Available from http://www.w3c.org/TR/xquery, 2002.

[8] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD), Montreal, Quebec, Canada*, pages 505–516, 4–6 June 1996. *SIGMOD Record* 25(2), June 1996.

[9] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for XML. In *Proc. of the 10th International World Wide Web Conference (WWW)Hong Kong, China*, pages 201–210, May 2001.

[10] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Reasoning about keys for XML. In *Proc. of the 8th Intl. Workshop on Data Base Programming Languages (DBPL), Frascati, Italy*, number 2397 in LNCS, Berlin, 2001. Springer-Verlag.

[11] Peter Buneman, Wenfei Fan, Jérôme Siméon, and Scott Weinstein. Constraints for semistructured data and XML. *SIGMOD Record*, 30:47–54, 2001.

[12] Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. To appear, 2002.

[13] L. Cardelli. Semistructured computation. In *Proc. of the Seventh Intl. Workshop on Data Base Programming Languages (DBPL)*, 1999.

[14] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998. Accepted for publication in *Theoretical Computer Science.*

[15] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. of Principles of Programming Languages (POPL).* ACM Press, January 2000.

[16] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proc. of Workshop on the Web and Data Bases (WebDB)*, 2000.

[17] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1998.

[18] G. Conforti, O. Ferrara, and G. Ghelli. TQL algebra and its implementation (extended abstract). In *Proc. of IFIP International Conference on Theoretical Computer Science (IFIP TCS), Montreal, Canada*, August 2002.

[19] G. Conforti, O. Ferrara, and G. Ghelli. TQL algebra and its implementation (full paper). To appear, 2002.

[20] A. Deutsch, D. Florescu M. Fernandez, A. Levy, and D. Suciu. A query language for XML. In *Proc. of the Eighth International World Wide Web Conference*, 1999.

[21] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a web-site management system. In *Proc. of Workshop on Management of Semistructured Data, Tucson*, 1997.

[22] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 414–425, 1998.

[23] Mary Fernandez, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages: Experiences and exemplars. Available from http://www-db.research.bell-labs.com/user/simeon/xquery.ps, 1999.

[24] G. Ghelli. TQL as an XML query language. Available from http://www.di.unipi.it/∼ghelli/papers.html, 2001.

[25] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Proc. of Workshop on the Web and Data Bases (WebDB)*, pages 25–30, 1999.

[26] 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.

[27] P. Kanellakis. Tutorial: Constraint programming and database languages. In *Proc. of the 14th Symposium on Principles of Database Systems (PODS), San Jose, California*, pages 46–53. ACM Press, 1995.

[28] G. Kuper, L. Libkin, and J. Paredaens. *Constraint Databases*. Springer-Verlag, Berlin, 2000.

[29] Y. Papakonstantinou, H.G. Molina, and J. Widom. Object exchange across heterogeneous information sources. *Proc. of the eleventh IEEE Int. Conference on Data Engineering, Birmingham, England*, pages 251–260, 1996.

A Proof of Lemma 4

Lemma 5 *Let $\mathcal{A}$ be a formula,* $\mathbf{V}$ *be a set of variables, let Ξ be a set $\{\xi_i\}^{i\in I}$ of recursion variables that includes those which are free in $\mathcal{A}$, and let γ be a function defined over Ξ such that, for every ξ_i, $\gamma(\xi_i)\in\mathcal{IT}\rightarrow\mathcal{T}^{\hat{\gamma}(\xi_i)}$ and $\hat{\gamma}(\xi_i)$ is disjoint from $\mathbf{V}$. then:*

$$\forall\rho\in\mathbf{1}^{\mathbf{V}}, I\in\mathcal{IT}.\quad \mathcal{B}(I,\mathcal{A})_{\rho,\gamma} \;=\; \{\rho' \mid \rho'\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})},\ I\in[\![\mathcal{A}]\!]_{(\rho;\rho'),\,d(\gamma,\rho')}\}$$

where

$$d(\gamma,\rho) \;=\; \lambda\xi\!:\!\Xi.\{I \mid \rho\in\gamma(\xi)(I)\}$$

Proof First observe that, even if $d(\gamma,\rho)$ is defined for every ρ, we only care about the result of its application to a ρ that belongs to $\mathbf{1}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})}$. We prove the theorem by induction on the size of $\mathcal{A}$. For simplicity, we focus on the case when $\mathbf{V}=\emptyset$, hence we prove that:

$$\forall I.\quad \mathcal{B}(I,\mathcal{A})_{\epsilon,\gamma} \;=\; \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\emptyset,\hat{\gamma})},\ I\in[\![\mathcal{A}]\!]_{\rho,\,d(\gamma,\rho)}\}$$

We first observe that for all $\mathcal{A}$, γ, the following statements are logically equivalent; the same is true if we reverse both $\subseteq$ and $\Rightarrow$, or if we consider set equality and $\Leftrightarrow$.

$$\begin{array}{lll}
\forall I & \mathcal{B}(I,\mathcal{A})_{\epsilon,\gamma} & \subseteq\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\emptyset,\hat{\gamma})},\ I\in[\![\mathcal{A}]\!]_{\rho,\,d(\gamma,\rho)}\}\\
\forall\rho\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\emptyset,\hat{\gamma})}, I & \rho\in\mathcal{B}(I,\mathcal{A})_{\epsilon,\gamma} & \Rightarrow\ I\in[\![\mathcal{A}]\!]_{\rho,\,d(\gamma,\rho)}\\
\forall\rho\in\mathbf{1}^{\mathcal{S}(\mathcal{A},\emptyset,\hat{\gamma})} & \{I \mid \rho\in\mathcal{B}(I,\mathcal{A})_{\epsilon,\gamma}\} & \subseteq\ [\![\mathcal{A}]\!]_{\rho,\,d(\gamma,\rho)}
\end{array}$$

We only consider the case when $\mathcal{A}=\mu\xi.\mathcal{C}$; the other cases are far easier.
We proof the equality by considering the two inclusions:

$$\begin{array}{l}
\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\ \subseteq\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}\\
\{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}\ \subseteq\ \mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}
\end{array}$$

We first prove that:

$$\forall\gamma,I.\ \mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\ \subseteq\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}$$

By definition of $\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}$, this means:

$$\begin{array}{l}
\forall\gamma,I.\ \mathit{Fix}(\lambda M:\mathcal{IT}\rightarrow\mathcal{P}(\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}).\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})(I)\\
\subseteq\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}
\end{array}$$

This is equivalent to:

$$\begin{array}{l}
\forall\gamma.\ \mathit{Fix}(\lambda M:\mathcal{IT}\rightarrow\mathcal{P}(\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}).\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})\\
\subseteq\ \lambda I.\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}
\end{array}$$

We prove it by showing that the r.h.s. is a fixpoint of the function $\lambda M.\lambda\mathcal{Y}.\ldots$:

$$\begin{array}{l}
\forall\gamma.\ (\lambda M.\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})(\lambda I.\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\})\\
=\ \lambda I.\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}
\end{array}$$

We apply β reduction; we will abbreviate $[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}$ as F_ρ:

$$\forall\gamma.\ \lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto\lambda I.\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},I\in F_\rho\}]}\ =\ \lambda I.\ \{\rho \mid \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in F_\rho\}$$

We reduce function comparison to pointwise comparison:

$$\forall\gamma, I.\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma[\xi\mapsto\lambda I.\ \{\rho\ |\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}, I\in F_\rho\}]} \ =\ \{\rho\ \mid\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in F_\rho\}$$

The definition of F_ρ (i.e., $[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}$) implies that $F_\rho = [\![\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)[\xi\mapsto F_\rho]}$:

$$\forall\gamma, I.\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma[\xi\mapsto\lambda I.\ \{\rho\ |\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}, I\in F_\rho\}]} \ =\ \{\rho\ \mid\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)[\xi\mapsto F_\rho]}\}$$

The proof follows immediately by induction, if we are able to prove that:

$$d(\gamma[\xi\mapsto\lambda I.\ \{\rho\ \mid\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in F_\rho\}],\rho)\ = d(\gamma,\rho)[\xi\mapsto F_\rho]$$

The two functions coincide for $\xi'\neq\xi$; when applied to ξ they yield:

$$\begin{aligned}&d(\gamma[\xi\mapsto\lambda I.\ \{\rho\ \mid\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in F_\rho\}],\rho)(\xi)\\ &=\ \{I\ \mid\ \rho\in\{\rho\ \mid\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in F_\rho\}\}\ =\ \{I\ \mid\ I\in F_\rho\}\ =\ F_\rho\ =\ (d(\gamma,\rho)[\xi\mapsto F_\rho])(\xi)\end{aligned}$$

We now prove the opposite inclusion:

$$\forall\gamma, I.\ \{\rho\ \mid\ \rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})},\ I\in[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\}\ \subseteq\ \mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}$$

This is equivalent to:

$$\forall\gamma,\rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}.\ [\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}\ \subseteq\ \{I\ \mid\ \rho\in\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\}$$

By the definition of $[\![\mu\xi.\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)}$, it is sufficient to prove that:

$$\forall\gamma,\rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}.\ [\![\mathcal{C}]\!]_{\rho,\,d(\gamma,\rho)[\xi\mapsto\{I\ |\ \rho\in\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\}]}\ \subseteq\ \{I\ \mid\ \rho\in\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\}$$

Let $\gamma'_{\mathcal{C},\gamma}=\gamma[\xi\mapsto\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mu\xi.\mathcal{C})_{\epsilon,\gamma}]$. Since $d(\gamma'_{\mathcal{C},\gamma},\rho)=d(\gamma,\rho)[\xi\mapsto\{I\ \mid\ \rho\in\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\}]$, then, the statement above can be rewritten as:

$$\forall\gamma,\rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}.\ [\![\mathcal{C}]\!]_{\rho,\,d(\gamma'_{\mathcal{C},\gamma},\rho)}\ \subseteq\ \{I\ \mid\ \rho\in\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\}$$

and this, by induction, is equivalent to:

$$\forall\gamma,\rho\in\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}.\ \{I\ \mid\ \rho\in\mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma'_{\mathcal{C},\gamma}}\}\ \subseteq\ \{I\ \mid\ \rho\in\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}\}$$

This can be deduced from:

$$\forall\gamma, I.\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma'_{\mathcal{C},\gamma}}\ \subseteq\ \mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}$$

We expand $\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}$:

$$\forall\gamma, I.\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma'_{\mathcal{C},\gamma}}\ \subseteq\ \mathit{Fix}(\lambda M:\mathcal{IT}\to\mathcal{P}(\mathbf{1}^{\mathcal{S}(\mu\xi.\mathcal{C},\emptyset,\hat{\gamma})}).\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})(I)$$

We unfold the fix point and apply it to I; we also expand $\gamma'_{\mathcal{C},\gamma}$:

$$\forall\gamma, I.\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma[\xi\mapsto\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mu\xi.\mathcal{C})_{\epsilon,\gamma}]}\ \subseteq\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma[\xi\mapsto \mathit{Fix}(\lambda M.\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})]}$$

We now replace $\mathcal{B}(I,\mu\xi.\mathcal{C})_{\epsilon,\gamma}$ at the left hand side with its definition.

$$\forall\gamma, I.\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma[\xi\mapsto\lambda\mathcal{Y}.\mathit{Fix}(\lambda M.\ \lambda\mathcal{Y}'.\mathcal{B}(\mathcal{Y}',\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})(\mathcal{Y})]}\ \subseteq\ \mathcal{B}(I,\mathcal{C})_{\epsilon,\gamma[\xi\mapsto \mathit{Fix}(\lambda M.\lambda\mathcal{Y}.\mathcal{B}(\mathcal{Y},\mathcal{C})_{\epsilon,\gamma[\xi\mapsto M]})]}$$

The two sides are equal. □