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

Luca Cardelli

Microsoft Research, Cambridge, UK

Giorgio Ghelli

Università di Pisa, Pisa, Italy

Received May 2002
Revised Apr 2003

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

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 (Abiteboul et al., 1999).

The ambient logic is a modal logic proposed to describe the structural and computational properties of distributed and mobile computation (Cardelli and Gordon, 2000). 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.

In this paper we present TQL, a query language for semistructured data that is based on the ambient logic.

The language turns out to be quite expressive, even though a TQL query is not much more than a nesting of comprehensions operations, each built around a logical formula expressed in our "tree logic". The fact that the tree logic can be naturally used to express types and constraints over semistructured data opens interesting possibilities. In a nutshell, problems like subtyping, constraint implication, constraint satisfiability, query correctness and query containment, become special cases of the validity problem for this logic. The same holds for their combinations, such as query containment in presence of constraints, or query correctness in presence of subtyping. The high level of expressiveness of the logic allows us to describe complex types and constraints. For example, the type and constraint languages proposed in (H. Hosoya, 2000) and (Buneman et al., 2001c) can be easily translated into the tree logic. 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, we believe that decidable subsets of the logic can be defined, which are expressive enough to encode interesting type and constraint systems. The search for decidable subsets with the "right" balance of expressiveness and cost is an open problem, but the first results in this field are emerging (Calcagno et al., 2003; Cohen, 2002). This unified framework for types, constraints, and queries is a central, but long-term, aim of the TQL project, and we are currently taking just the first steps in this direction.

In this paper we first introduce the query language TQL through some examples, then we present its full formal definition, and finally we define a formal execution model that is the basis of the current TQL implementation.

The paper is structured as follows. In Section 2 we 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 of Section 5 is based. In Section 6 we briefly discuss how to represent 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 (W3C, 2002a). 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[ProcICDT99]
    | 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 intended as a language to query semistructured data in general, i.e. unordered trees with labeled edges; XML is just one way to construct such trees, using tagged elements (and attributes) to build labeled 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 result of the query.

For example, consider the following TQL query, which 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 logical 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 we shall call `$t`". Since `//tql.di.unipi.it/tql/pubb.xml` contains two books with year 1999, and with titles `DataOnTheWeb` and `ProcICDT99`, the query first computes the set of bindings:

```
{[$t = DataOnTheWeb]; [$t = ProcICDT99]}
```

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

```
title[DataOnTheWeb] | title[ProcICDT99].
```

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 $\mathtt{A}_1$ | $\mathtt{A}_2$, 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 analyze the horizontal structure of a tree. For example, in the next query, by matching the formula `year[1999] | $EverythingElse` against each book, we return, for any book whose year is 1999, everything but the year. Here `.a.b[A]` abbreviates `.a[.b[A]]`:

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

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

```
BookOf1999[ title[DataOnTheWeb] | author[ first[Serge] | last[Abiteboul] ] ... ]
| BookOf1999[ title[ProcICDT99] | editor[ first[Peter] | last[Buneman] ] ... ]
```

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

For example, the following query returns any tag inside a `book` tagging 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]`, $\texttt{A}_1 \mid \texttt{A}_2$, ...) can be used to express matching conditions, and the others can be used to combine such conditions, and to quantify variables.

For example, the condition in the following query requires the existence of a `title` field, of a `$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]`, which will be used hereafter ('`%`' can be read as 'match any label').

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 another derived path operator, `.%*[A]`, that matches a path of arbitrary length (this is formally described in Section 4.3).

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

```
from $Bib |= .%*.$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

† In `Exists $x. .$x[Springer]` we have two dots: the first belongs to `Exists` (as in $\exists x.P(x)$), while the second belongs to `.$x[Springer]`, and means `$x[Springer] | T`.

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 in 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 `$A` that satisfy the outer condition, while the inner formula, which is evaluated once for each binding `[$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 closed formula `A` expresses a property of interest, we can check it by running a query like `from Q |= A select success`: this query returns an edge labeled `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 formulas are negated. In fact, when negation appears in a query, in most cases the TQL optimizer pushes it down to the atomic formulas (tree variables, tree emptiness, 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 express 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 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 it is impossible to find two separate books with the same title (more precisely, with one title in common). As above, the | operator can be used to express the fact that we have two distinct subtrees with the same title. In traditional logical approaches, based on first order or modal logics, we need some notion of "node identity", or "world identifier", in order to express the existence of two distinct nodes that satisfy a given property.

```
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 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 labeled `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*

In a TQL query a tag variable can appear wherever a tag can appear. Hence, we can take the query that checks whether `title` is a key, and substitute `title` with `$k`, as follows:

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

This query is well formed, and it returns the set of all subtags of book whose content is a key for our set of books.

This is an instance of a general property of TQL. For every query Q that checks a property P of a tag `t`, if we substitute `t` with a tag variable, we obtain a query that finds the set of *all* tags that satisfy P. And if this set is finite, our implementation will compute it.

This unique property is due to the fact that TQL does not constraint the appearance of free variables in formulas. For example, 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 over an open formula is allowed. The query evaluation algorithm we exploit to allow this kind of quantification is indeed non standard, and quite sophisticated. It is described in (Conforti et al., 2003).

In the other query languages, this kind of generalization is definitely less trivial. 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 generalization 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). The query extracts all the tags in the database and lists, for each one, all the labels that must or may appear, and distinguishes among those the ones that may be repeated and the ones that appear only once. This query may look frightening, at a first sight, but it is just a 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 computes two partitions of its subtags, the first that divides mandatory from optional tags, and the second that divides list (i.e., repeatable) vs. non-list 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 sometimes appears once (`.%*.$tag[ .$subtag[T]]`) but it never appears twice: `Not .%*.$tag[ .$subtag[T] | .$subtag[T] ]`.

Here is the query, defined over a database `$parts`.

```
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 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.

All the queries in this section, as written here, have been checked on the TQL implementation. Running such queries on realistic pieces of data requires, in our prototype, quite a long time. This is not surprising, since the current implementation is a 'proof of concept', aimed at showing that such a language can be implemented. Much work remains to be done on query optimization.

We hope that the reader is now curious 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. The syntax, and in a sense the semantics, of information trees corresponds to the 'spatial' subset of the ambient calculus, i.e. to ambients with no actions (Cardelli and Gordon, 2000).

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 simplify some aspects of the logic, but in this paper we stick to the original notion of unordered trees from (Cardelli and Gordon, 2000).

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 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 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 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 (Cardelli and Gordon, 1998). We define a function $[\![F]\!]$ mapping the info-term F to the denoted information tree.

Table 3.1. *Info-terms and their information tree meaning*

$F ::=$	info-term			
$\mathbf{0}$	denoting the empty multiset			
$m[F]$	denoting the multiset $\{\langle m, F\rangle\}$, where $m \in \Lambda$			
$F \mid F$	denoting multiset union			
$[\![\mathbf{0}]\!]$	$=_{def}$ $\mathbf{0}$		$=_{def}$	$\{\}$
$[\![m[F]]\!]$	$=_{def}$ $m[[\![F]\!]]$		$=_{def}$	$\{\langle m, [\![F]\!]\rangle\}$
$[\![F' \mid F'']\!]$	$=_{def}$ $[\![F']\!] \mid [\![F'']\!]$		$=_{def}$	$[\![F']\!] \uplus [\![F'']\!]$

We often abbreviate $m[\mathbf{0}]$ as $m[]$, or as m. We assume that Λ includes the disjoint union of any 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. It coincides with ambient-calculus congruence, when restricted to this set of terms. This relation is called *info-term congruence*, and it can be axiomatized as the minimal congruence that includes the commutative monoidal laws for $\mid$ and $\mathbf{0}$, as follows.

Table 3.2. *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 (Papakonstantinou et al., 1996), UnQL trees (Buneman et al., 1996), and XML Query Data Model (W3C, 2002c). 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. Our approach can be applied to graphs as well, by substituting the tree-edge constructor $m[F]$ with a graph-edge constructor $\mathit{label}(\mathit{fromNode}, \mathit{toNode})$, and the tree logic with the corresponding graph logic defined in (Cardelli et al., 2002). However, we believe that a full graph language would also need operators to create new nodes and to hide the identity of nodes. For this reason, we prefer to focus here on the simpler issue of trees, which is rich enough to warrant a separate study, and we leave the issues of node hiding and generation to future studies (Cardelli et al., 2003).

UnQL trees are characterized by the fact that they are considered equivalent 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 (W3C, 2002c). 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-labeled forests, while we talk in terms of edge-labeled trees. The two are perfectly isomorphic. TQL data can be seen as node-labeled 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 labeled 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 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 (Cardelli and Gordon, 2000). The ambient logic is particularly attractive for us because it is equipped with a large set of logical laws for tree-like structures, in particular logical equivalences, which provide a foundation for query rewriting rules and query optimization. Moreover, we hope to exploit the current research on decision procedures for (sublogics of) the ambient logic (Calcagno et al., 2003; Cohen, 2002), to build tools to decide the problems (query correctness, containment, equivalence) that we described in Section 1.

We start here from a subset of the ambient logic as presented in (Cardelli and Gordon, 2000), but we enrich it with information tree variables, label comparison, and recursion. All the results in Sections 4.1 and 4.2 are standard results of the ambient logic transposed to this specific variant. For this reason we do not detail here the proofs, but give only the essential outline.

4.1. *Formulas*

The syntax of 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 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.

We assume that the quantifiers $\exists x.\mathcal{A}$, $\exists \mathcal{X}.\mathcal{A}$, and $\mu\xi.\mathcal{A}$, bind their variables as far to the right as possible; for example, $\exists x.\mathcal{A} \wedge \mathcal{A}'$ means $\exists x.(\mathcal{A} \wedge \mathcal{A}')$. Negation binds more strongly than any other operator, so that $\neg\mathcal{A} \wedge \mathcal{A}'$ means $(\neg\mathcal{A}) \wedge \mathcal{A}'$. No other precedence rule is assumed.

Table 4.1. *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.

Table 4.2. *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 $\emptyset$
$[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta}$	$=_{def}$	$\bigcap \{S \subseteq \mathcal{IT} \mid S \supseteq [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]}\}$
$[\![\xi]\!]_{\rho,\,\delta}$	$=_{def}$	$\delta(\xi)$

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 $F \vDash_{\rho,\delta} \mathcal{A}$:

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

We also talk about information trees satisfying a formula, as follows:

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

The context will disambiguate the notation. In both cases we omit δ when it is the empty function.

The semantic definition is probably easier to understand in terms of the associated satisfaction relation. For example, the interpretation of $\exists$ corresponds to the following property of the satisfaction relation:

$$\begin{array}{lll} F \vDash_{\rho,\delta} \exists\mathcal{X}.\mathcal{A} & \Leftrightarrow_{def} & [\![F]\!] \in \bigcup_{I\in\mathcal{IT}} [\![\mathcal{A}]\!]_{\rho[\mathcal{X}\mapsto I],\,\delta} \\ & \Leftrightarrow & \exists I \in \mathcal{IT}.\ [\![F]\!] \in [\![\mathcal{A}]\!]_{\rho[\mathcal{X}\mapsto I],\,\delta} \\ & \Leftrightarrow & \exists I \in \mathcal{IT}.\ F \vDash_{\rho[\mathcal{X}\mapsto I],\delta} \mathcal{A} \end{array}$$

Along the same lines, one can prove the following properties of conjunction and negation:

$$\begin{array}{lcl} F \vDash_{\rho,\delta} \neg\mathcal{A} & \Leftrightarrow & \neg(F \vDash_{\rho,\delta} \mathcal{A}) \\ F \vDash_{\rho,\delta} \mathcal{A} \wedge \mathcal{B} & \Leftrightarrow & F \vDash_{\rho,\delta} \mathcal{A} \;\wedge\; F \vDash_{\rho,\delta} \mathcal{B} \end{array}$$

The $\mid$ case is characterized by the following property:

$$\begin{array}{ll} & F \vDash_{\rho,\delta} \mathcal{A} \mid \mathcal{B} \\ \Leftrightarrow_{def} & \exists I', I''.\ [\![F]\!] = I' \mid I'',\ I' \in [\![\mathcal{A}]\!]_{\rho,\,\delta},\ I'' \in [\![\mathcal{B}]\!]_{\rho,\,\delta} \\ \Leftrightarrow & \exists I', I'', F', F''.\ [\![F]\!] = I' \mid I'',\ I' = [\![F']\!],\ I'' = [\![F'']\!],\ F' \vDash_{\rho,\delta} \mathcal{A},\ F'' \vDash_{\rho,\delta} \mathcal{B} \\ \Leftrightarrow & \exists F', F''.\ [\![F]\!] = [\![F' \mid F'']\!],\ F' \vDash_{\rho,\delta} \mathcal{A},\ F'' \vDash_{\rho,\delta} \mathcal{B} \\ \Leftrightarrow & \exists F', F''.\ F \equiv F' \mid F'',\ F' \vDash_{\rho,\delta} \mathcal{A},\ F'' \vDash_{\rho,\delta} \mathcal{B}. \end{array}$$

We list the essential property of each operator in Table 4.3 below. One may use these properties as a definition of the satisfaction relation, as done in the original ambient-logic paper (Cardelli and Gordon, 2000); we follow here the style of (Caires and Cardelli, 2003), because it works better with the recursion operator.

The semantics of $\mu\xi.\mathcal{A}$ is defined here as the least fixpoint of a function that maps a set of trees S to a set of trees $[\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]}$. The definition above is actually formulated in terms of the least ($\bigcap$) pre-fixpoint, which coincides, by standard lattice-theory arguments, with the least fixpoint of $\lambda S.\ [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]}$, since that function is monotone in S (Lemma 2). This definition of the semantics induces, on the satisfaction relation, the following property: $F \vDash_{\rho,\delta} \mu\xi.\mathcal{A} \ \Leftrightarrow\ F \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\}$ (Lemma 3).

The valuation ρ is 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 describing the process of finding all possible answers to a query with respect to a database I.

Table 4.3. *Some properties of satisfaction*

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

Most of the properties in Table 4.3 are easy to prove. For the recursive case, we need a couple of lemmas.

Lemma 1 (Substitution).

$$[\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto[\![\mathcal{A}']\!]_{\rho,\,\delta}]} = [\![\mathcal{A}\{\xi \leftarrow \mathcal{A}'\}]\!]_{\rho,\,\delta}$$

Lemma 2 (Monotonicity). For any A, if ξ appears only positively in A, then

$$S \subseteq S' \Rightarrow [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]} \subseteq [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S']}.$$

If ξ appears only negatively, then

$$S \subseteq S' \Rightarrow [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]} \supseteq [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S']}.$$

Lemma 3 (Properties of Satisfaction). The properties of Table 4.3 hold.

Proof. A few cases are proved in the text before the table; the others are trivial, apart from the recursive case.

For the property

$$F \vDash_{\rho,\delta} \mu\xi.\mathcal{A} \Leftrightarrow F \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\},$$

we first observe that $\lambda S.\ [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]}$ is monotone by Lemma 2, since ξ only appears positively in $\mathcal{A}$. Hence, by Knaster-Tarski lemma, $[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta}$ is a fixpoint of that function, i.e. (1): $[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta} = [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta}]}$. The thesis now follows:

$$\begin{array}{lcll} F \vDash_{\rho,\delta} \mu\xi.\mathcal{A} & \Leftrightarrow & \text{(By def.)} & [\![F]\!] \in [\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta} \\ & \Leftrightarrow & \text{(By 1)} & [\![F]\!] \in [\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta}]} \\ & \Leftrightarrow & \text{(By Lemma 1)} & [\![F]\!] \in [\![\mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\}]\!]_{\rho,\,\delta} \\ & \Leftrightarrow & \text{(By def.)} & F \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\} \end{array}$$

□

4.2. *Some Derived Operators*

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

Table 4.4. *Derived Operators*

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

$F \vDash m[\Rightarrow \mathcal{A}]$ 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: $F \vDash \mathit{book}[.\mathit{author}[\mathit{Date}]]$
— If F is a book, then *Date* is an author: $F \vDash \mathit{book}[\Rightarrow .\mathit{author}[\mathit{Date}]]$

$F \vDash \mathcal{A} \mid\mid \mathcal{B}$ 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 | and the || operators, consider the following statements.

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

The dual of the least fixpoint operator $\mu\xi.\mathcal{A}$ is the greatest fixpoint operator $\nu\xi.\mathcal{A}$; this operator is not very useful in the present context, since we only use TQL to query finite trees. For example, on finite trees, both $\mu\xi.\mathbf{0} \vee m[\xi]$ and $\nu\xi.\mathbf{0} \vee m[\xi]$ describe every information tree that matches $m[m[\ldots m[\mathbf{0}]\ldots]]$. However, the infinite tree $m[m[\ldots]]$ is only matched by $\nu\xi.\mathbf{0} \vee m[\xi]$.

Satisfaction over the derived operators enjoys the following properties. The first two are obvious, while the next two are more subtle, and include a coinduction principle.

Again, these properties form the basis for a pattern matching algorithm. We omit the obvious properties of **F**, disjunction, and universal quantification.

Table 4.5. *Some properties of satisfaction for derived operators*

$$
\begin{array}{l}
F \vDash_{\rho,\delta} \eta[\Rightarrow \mathcal{A}] \Leftrightarrow\ \forall F'.\ (F \equiv \rho(\eta)[F']\ \Rightarrow F' \vDash_{\rho,\delta} \mathcal{A}) \\
F \vDash_{\rho,\delta} \mathcal{A} \mid\mid \mathcal{B}\ \Leftrightarrow\ \forall F', F''.\ F \equiv F' \mid F''\ \Rightarrow\ (F' \vDash_{\rho,\delta} \mathcal{A}\ \vee\ F'' \vDash_{\rho,\delta} \mathcal{B}) \\
F \vDash_{\rho,\delta} \nu\xi.\mathcal{A}\ \Leftrightarrow\ F \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \nu\xi.\mathcal{A}\} \\
F \vDash_{\rho,\delta} \nu\xi.\mathcal{A}\ \Leftrightarrow\ \exists\mathcal{B}.\ F \vDash_{\rho,\delta} \mathcal{B}\ \wedge\ \forall F'.\ F' \vDash_{\rho,\delta} \mathcal{B} \Rightarrow F' \vDash_{\rho,\delta} \mathcal{A}\{\xi \leftarrow \mathcal{B}\}
\end{array}
$$

Many logical equivalences have been derived for the ambient logic, and are inherited by the tree logic. These equivalences can be exploited by a query logical optimizer. For example, the properties we list below can be used to reduce the size of the formula to be evaluated; the first six may generate a **F**/**T**, the last six would propagate it. More equations are listed in Appendix B.

Table 4.6. *Some equations*

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

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 we 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 we 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.

Table 4.7. *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:

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

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 we 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 $|$.

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

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.

Table 5.1. *TQL queries*

$Q ::=$	query
from $Q \vDash \mathcal{A}$ *select* Q'	valuation-collecting query
$\mathcal{X}$	matching variable
$\mathbf{0}$	empty result
$Q \mid Q$	composition of results
$\eta[Q]$	nesting of result
$f(Q)$	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 $\mathcal{IT} \to \mathcal{IT}$, 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;
— *sum(I)*, yielding $n[\mathbf{0}]$, where n is the sum of (the multiset of) all the integers i such that $i[\ldots]$ appears in I.

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

The definition of *free variables* in a query is standard, except for the *from* $Q \vDash \mathcal{A}$ *select* Q' 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 binds in Q' all variables that are free in $\mathcal{A}$; this is formalized in the first line in the following table.

Table 5.2. *Free variables in TQL queries*

$FV($*from* $Q \vDash \mathcal{A}$ *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. finite domain functions mapping each $\mathcal{V}_i \in \mathbf{V}$ to an element of $U(\mathcal{V}_i)$.

The semantics of a query is defined with respect to a 'context valuation' $\rho^{\mathbf{V}}$, that binds all the variables that occur free in the query. This context valuation is used to bind some top-level names, like `$Bib` in Section 2, to the documents to be queried. Moreover, in a query *from* $Q \vDash \mathcal{A}$ *select* Q', the binder $Q \vDash \mathcal{A}$ generates the context valuations that will be used to evaluate Q', by enriching the current context valuation with values for the variables in $FV(\mathcal{A})$.

The semantics of a binder and of a query are defined in the following table.

A binder $Q \vDash \mathcal{A}$ denotes a function that takes one valuation $\rho^{\mathbf{V}}$ such that $\mathbf{V} \supseteq FV(Q)$ and returns a set of valuations $[\![Q \vDash \mathcal{A}]\!]_{\rho^{\mathbf{V}}}$. More precisely, it returns all valuations $\rho'^{\mathbf{V}'}$ that extend the context valuation $\rho^{\mathbf{V}}$ and such that $[\![Q]\!]_{\rho^{\mathbf{V}}} \vDash_{\rho'^{\mathbf{V}'}} \mathcal{A}$. The notation $\rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}}$ means that the graph of the function $\rho'^{\mathbf{V}'}$ includes that of $\rho^{\mathbf{V}}$. This means that $\mathbf{V}' \supseteq \mathbf{V}$ and that $\rho'^{\mathbf{V}'}$ and $\rho^{\mathbf{V}}$ coincide over $\mathbf{V}$, i.e. the new valuations do not change the already defined variables, but assign values to the other free variables.

A query Q denotes a function that takes a valuation $\rho^{\mathbf{V}}$ such that $\mathbf{V} \supseteq FV(Q)$ and returns a tree $[\![Q]\!]_{\rho^{\mathbf{V}}}$. A query *from* $Q \vDash \mathcal{A}$ *select* Q' is evaluated by evaluating the subquery Q' once for each valuation ρ' that is computed by the binder; all the resulting trees $[\![Q']\!]_{\rho'}$ are then combined using *Par*, the n-ary version of the binary operator |, defined in Section 3.

Table 5.3. *Query semantics*

$[\![Q \vDash \mathcal{A}]\!]_{\rho^{\mathbf{V}}}$	$= \{\rho'^{\mathbf{V}'} \mid \mathbf{V}' = \mathbf{V} \cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ [\![Q]\!]_{\rho^{\mathbf{V}}} \vDash_{\rho'^{\mathbf{V}'}} \mathcal{A}\}$
$[\![\mathcal{X}]\!]_{\rho^{\mathbf{V}}}$	$= \rho^{\mathbf{V}}(\mathcal{X})$
$[\![\mathbf{0}]\!]_{\rho^{\mathbf{V}}}$	$= \mathbf{0}$
$[\![Q \mid Q']\!]_{\rho^{\mathbf{V}}}$	$= [\![Q]\!]_{\rho^{\mathbf{V}}} \mid [\![Q']\!]_{\rho^{\mathbf{V}}}$
$[\![m[Q]]\!]_{\rho^{\mathbf{V}}}$	$= m[[\![Q]\!]_{\rho^{\mathbf{V}}}]$
$[\![x[Q]]\!]_{\rho^{\mathbf{V}}}$	$= \rho^{\mathbf{V}}(x)[[\![Q]\!]_{\rho^{\mathbf{V}}}]$

$$\begin{array}{ll} [\![f(Q)]\!]_{\rho \mathbf{v}} & = f([\![Q]\!]_{\rho \mathbf{v}}) \\ [\![\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q']\!]_{\rho \mathbf{v}} & = \mathit{Par}_{\rho' \mathbf{v}' \in [\![Q \vDash \mathcal{A}]\!]_{\rho \mathbf{v}}}\ [\![Q']\!]_{\rho' \mathbf{v}'} \end{array}$$

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; the static-check option is discussed in Section 5.4. The current implementation of TQL executes a run-time test that, whenever $[\![Q \vDash \mathcal{A}]\!]_{\rho \mathbf{v}}$ is infinite, raises an 'infinite result' run-time exception. We discuss this theme in the next two subsections.

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:

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

Intuitively, every entry in DB that is an *author* binds $\mathcal{X}$ but not $\mathcal{Y}$, and vice-versa for *autore* entries. Formally, an unbound variable corresponds to an infinite amount of valuations; for example, if $\rho(DB) = \mathit{author}[m[]]$, then $[\![DB \vDash (\mathit{author}[\mathcal{X}] \vee \mathit{autore}[\mathcal{Y}]) \mid \mathbf{T}]\!]_\rho$ is the infinite set of triples:

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

Negation creates a similar problem. Consider the following query.

$$\begin{array}{l} \mathit{from}\ \ DB \vDash \neg \mathit{author}[\mathcal{X}] \\ \mathit{select}\ \ \mathit{notAuthor}[\mathcal{X}] \end{array}$$

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

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

and the query has the following infinite result:

$$\mathit{Par}_{I \in (\mathcal{IT} \setminus \{m[]\})}\ \mathit{notAuthor}[I]$$

Some queries generate either a finite or an infinite tree, depending on the context valuation. For example, if $\mathcal{A}$ is a closed formula, then we have:

$$\begin{array}{rcl} [\![DB \vDash \mathcal{A} \wedge \neg\mathcal{X}]\!]_\rho & = & \{\rho'\ \mid\ \rho' \supseteq \rho,\ \rho(DB) \vDash_{\rho'} \mathcal{A} \wedge \neg\mathcal{X}\ \} \\ & = & \{\rho'\ \mid\ \rho' \supseteq \rho,\ \rho(DB) \vDash \mathcal{A},\ \rho'(\mathcal{X}) \neq \rho(DB)\ \} \\ & = & \begin{cases} \emptyset & \text{if} \quad \neg\rho(DB) \vDash \mathcal{A} \\ \{\rho'\ \mid\ \rho' \supseteq \rho,\ \rho'(\mathcal{X}) \neq \rho(DB)\ \} & \text{if} \quad \rho(DB) \vDash \mathcal{A} \end{cases} \end{array}$$

Hence the query

$$\mathit{from}\ \ DB \vDash \mathcal{A} \wedge \neg\mathcal{X}\ \mathit{select}\ \ a[\mathcal{X}]$$

returns an infinite tree if $\rho(DB) \vDash \mathcal{A}$, and the empty tree otherwise.

We say that a query is safe when its semantics is always finite, independently of the context valuation. We say that a formula $\mathcal{A}$ is safe with respect to a set of bound variables $\mathbf{V}$ when, for each $\mathcal{X} \notin (\mathbf{V} \cup FV(\mathcal{A}))$ and for each valuation ρ for $\mathbf{V} \cup \mathcal{X}$, its semantics $[\![\mathcal{X} \vDash \mathcal{A}]\!]_\rho$ 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.

Formula and query safety are undecidable. Consider again the formula $\mathcal{A} \wedge \neg\mathcal{X}$; it generates an infinite set of bindings if and only if is applied to a database I such that $I \vDash_\epsilon A$. Hence, it is safe iff $\mathcal{A}$ is unsatisfiable, i.e. $\neg\mathcal{A}$ is valid. But validity is undecidable for the tree logic (Charatonik and Talbot, 2001).

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. For example, the formula $((\neg \mathit{autore}[\mathcal{X}]) \wedge \mathit{author}[\mathcal{X}])$ is safe but the first conjunct is not. The formula $\forall x.\ y[\neg(x[\mathbf{T}])]$ is safe, but the subformula inside the quantifier is not.

This problem has been traditionally confronted by defining a decidable subclass of 'hereditarily finite' queries with the property that both the query and all its subqueries yield finite results. Then one defines a larger, and still decidable, class of queries that can easily be rewritten in this 'hereditarily finite' form. Queries in this larger class are evaluated, while every other query is discarded as 'not statically safe' (Ullman, 1982; Gelder and Topor, 1991; Abiteboul et al., 1995).

This approach is not very satisfactory, since many safe queries have to be discarded, and because complex syntactic conditions have to be chosen, in order to capture a large enough class of queries. The main advantage of this traditional approach, when applied to relational queries, is that the static-safeness conditions can be chosen so that the allowed queries can be translated into an efficient algebraic expression. We are not interested into this aspect since we want here to search for a new tree-relational algebra better suited for tree query languages, rather than studying the translation of TQL to traditional relational algebras.

For these reasons, we pursue here a different road: we define an evaluation mechanism that works with every formula, safe or unsafe. The mechanism is based on a finite representation of every computed set of valuations, finite or infinite. This mechanism allows us to evaluate every binder as it is, with no need to discard some as unsafe or to rewrite others to a more acceptable form. In this way, our optimizer is free to rewrite any formula into any other formula, without worrying about syntactic-safety conditions. This approach is not new in the database field; it can be described as a generalization of the constraint database approach (Kanellakis, 1995; Kuper et al., 2000).

When the top-down evaluation of a binder is completed, the final result may be either finite or infinite, even if the intermediate results were infinite. At this point, if the computed set of bindings R is infinite, we raise a run-time error, since we are not interested,

in the current implementation, in defining a finite representation of the infinite tree that would result if we used R to evaluate the *select* branch.

Hence, although we have solved the problem of evaluating unsafe binders, we have still a reason to try and statically identify a class of formulas which are guaranteed never to return an infinite set of valuations: this would allow us to statically analyze a query and tell the programmer that it is guaranteed never to raise an 'infinite-result' exception. However, this is quite different from relational safety tests. In that case, only the formulas that pass the test are translated into the algebra and executed. In our case, every formula can be translated and executed, but, if it did not pass the test, we know that it may return an infinite result.

The formula evaluation mechanism that copes with infinite results is described in Section 7, and an example of a static analysis algorithm to characterize a decidable subset of the safe formulas is described in Section 5.4.

A different solution, widely studied in the database literature, is to modify the semantics so that each query is evaluated using, instead of Λ, the *active domain*, i.e. the subset of Λ that only contains the labels found in the database and in the query. In this way, only finite results can be generated.

This approach is not satisfactory since it makes the semantics of a query depend on the constants appearing in parts of the database that may be completely unrelated to the query itself; for this reason, we will not consider it here. Actually, the active domain semantics is mostly advocated as a tool for theoretic studies about the expressive power of different query languages, or as a tool in the study of 'domain independent queries', i.e. queries whose semantics does *not* depend on the set Λ. For a discussion of the classical problem of domain independent queries, and for more references, the reader may consult any database textbook, such as (Abiteboul et al., 1995; Ullman, 1988).

5.4. *Restricted queries*

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

We define a predicate $\mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V}$ ($\mathcal{A}$ binds $\mathcal{V}$ if $\mathbf{V}$ is bound) that implies, informally, that for any I and for any valuation for $\mathbf{V}$, $\mathcal{A}$ can only extract finitely many matches from I; (see Property 1 below for a formal definition). The set of already-bound variables $\mathbf{V}$ is only used when dealing with the equality operator.

The binding predicate is defined as follows. For simplicity, we assume that negation is pushed down to the leaves of the formula. We have no rules for $\neg\mathcal{X}$, $\mathbf{T}$, and $\eta[\Rightarrow \mathcal{A}]$, since these formulas do not bind any variable. We omit the symmetric rules ($y = x$, $n = x$) for the equality case.

Observe that, as specified by Property 1, the predicate only computes a decidable

approximation of the semantic binding relation *sembinds*:

$$\begin{array}{l} \mathit{sembinds}(\mathbf{V}, \mathcal{A}, \mathcal{V}) \\ \quad \Leftrightarrow_{def} \ \forall I,\, \rho^{\mathbf{V}},\, \mathbf{V}' \supseteq \mathbf{V} \cup FV(\mathcal{A}) \cup \{\mathcal{V}\}.\ \{\rho'^{\mathbf{V}'}(\mathcal{V}) \mid \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ I \vDash_{\rho'^{\mathbf{V}'}} \mathcal{A}\} \text{ is finite} \\ \text{Property 1: } \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \Rightarrow \mathit{sembinds}(\mathbf{V}, \mathcal{A}, \mathcal{V}) \end{array}$$

For example, the closed formula $\mathcal{A} = \forall \mathcal{X}.\ \mathcal{X} \wedge \neg \mathcal{X}$, binds every variable, since it is unsatisfiable and hence $\{\rho'^{\mathbf{V}'}(\mathcal{V}) \mid I \vDash_{\rho'^{\mathbf{V}'}} \mathcal{A}\}$ is empty, while our predicate does not prove $\mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V}$ for any $\mathcal{V}$. We may perform a better analysis, but the true binding relation is in general undecidable, since it is at least as hard as unsatisfiability of closed formulas, as discussed before.

Table 5.4. *The binding predicate*

$\mathbf{V} \vdash \mathbf{F} \triangleright \mathcal{V}$	for any $\mathcal{V}$	$\mathbf{V} \vdash \mathcal{X} \triangleright \mathcal{X}$
$\mathbf{V} \vdash (x = y) \triangleright x \ \Leftrightarrow y \in \mathbf{V}$		$\mathbf{V} \vdash (x = n) \triangleright x$
$\mathbf{V} \vdash (\mathcal{A} \wedge \mathcal{B}) \triangleright \mathcal{V} \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \vee \mathbf{V} \vdash \mathcal{B} \triangleright \mathcal{V}$		$\mathbf{V} \vdash (\mathcal{A} \vee \mathcal{B}) \triangleright \mathcal{V} \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \wedge \mathbf{V} \vdash \mathcal{B} \triangleright \mathcal{V}$
$\mathbf{V} \vdash (\mathcal{A} \mid \mathcal{B}) \triangleright \mathcal{V} \ \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \vee \mathbf{V} \vdash \mathcal{B} \triangleright \mathcal{V}$		$\mathbf{V} \vdash (\mathcal{A} \mid\mid \mathcal{B}) \triangleright \mathcal{V} \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \wedge \mathbf{V} \vdash \mathcal{B} \triangleright \mathcal{V}$
$\mathbf{V} \vdash \eta[\mathcal{A}] \triangleright \mathcal{V} \quad \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \vee \eta = \mathcal{V}$		
$\mathbf{V} \vdash (\forall \mathcal{V}'.\ \mathcal{A}) \triangleright \mathcal{V} \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \wedge \mathcal{V}' \neq \mathcal{V}$		$\mathbf{V} \vdash (\exists \mathcal{V}'.\ \mathcal{A}) \triangleright \mathcal{V} \Leftrightarrow \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V} \wedge \mathcal{V}' \neq \mathcal{V}$

Now we can talk about query safety. A query Q is safe with respect to a context valuation $\rho^{\mathbf{V}}$ if $\mathit{safe}(Q, \mathbf{V})$ holds. $\mathit{safe}(\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q', \mathbf{V})$ holds if: all free variables in $\mathcal{A}$ are bound by $\mathcal{A}$, 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.

Table 5.5. *Query safety with respect to a context substitution that binds* $\mathbf{V}$

$\mathit{binds}(\mathbf{V}, \mathcal{A}) =_{def} \mathbf{V} \cup \{\mathcal{V} \mid \mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V}\}$	
$\mathit{safe}(\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q', \mathbf{V})$	$\Leftrightarrow FV(\mathcal{A}) \subseteq \mathit{binds}(\mathbf{V}, \mathcal{A}) \wedge \mathit{safe}(Q, \mathbf{V}) \wedge \mathit{safe}(Q', \mathit{binds}(\mathbf{V}, \mathcal{A}))$
$\mathit{safe}(\mathcal{X}, \mathbf{V})$	
$\mathit{safe}(\mathbf{0}, \mathbf{V})$	
$\mathit{safe}(Q \mid Q', \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V}) \wedge \mathit{safe}(Q', \mathbf{V})$
$\mathit{safe}(m[Q], \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V})$
$\mathit{safe}(x[Q], \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V})$
$\mathit{safe}(f(Q), \mathbf{V})$	$\Leftrightarrow \mathit{safe}(Q, \mathbf{V})$

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:

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

Property 2. If $\rho^{\mathbf{V}}$ ranges over valuations mapping the variables in $\mathbf{V}$ to labels or to

finite trees, then:

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

The relation $\mathbf{V} \vdash \mathcal{A} \triangleright \mathcal{V}$ is very similar to the relations that are used to define static notions of safety for relational calculus queries, but we are more liberal with respect to quantified variables. For example, in (Gelder and Topor, 1991), a query is considered *allowed* when (a) all free variables are bound and (b) for every quantification $\exists x.\mathcal{A}$, x is bound by $\mathcal{A}$ (other authors embed (b) condition in the definition of the binding relation (Abiteboul et al., 1995)). We only require (a). ‡

This difference derives from the fact that the classical notion of static safety is meant to prove that the query is finite *and* can be translated to an algebraic query defined over finite relations. We are interested in the finiteness of the query result, but we are going to translate it into an algebra of infinite tables, where we are able to implement quantification over infinite structures, hence we do not need to require that quantified variables are bound.

6. TQL Logic, Schemas, and Constraints

6.1. *Schemas*

Traditional path-based query languages explore the vertical structure of trees. Our logic can easily describe the horizontal structure as well, as is common in schemas for semistructured data. (E.g. in XML DTDs, XDuce Types (H. Hosoya, 2000), and XSD Schemas (W3C, 2002b); 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 (element of) $\mathcal{A}$ next to an (element of) $\mathcal{B}$
$\mathcal{A} \vee \mathcal{B}$	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.

However, such proposals usually come with an efficient algorithm for checking membership or other properties. For example, an efficient algorithm to check subtyping for XDuce types, based on a set-inclusion constraint solver, is presented in (Hosoya et al., 2000). These efficient algorithms, of course, do not fall out automatically from a general framework such as ours.

‡ Van Gelder-Topor analysis is actually more sophisticated; we are simplifying it for ease of comparison.

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 keys have been studied for semistructured data. For example, Buneman et al. (Buneman et al., 2001a) define a notion of *relative* keys. Consider a set of books whose type, expressed as in the previous section, is:

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

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 $\mathcal{Y}$:

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

The notion actually defined in (Buneman et al., 2001a) 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 verbose to express in first-order logic, especially because the actual definition of (Buneman et al., 2001a) must distinguish between node-equality, used to compare e' and e'', from value-equality, used to compare their P_i-reachable subelements.

When TQL logic is used, instead of first-order logic, the same notion becomes much easier to formalize. It is fully captured by the following formula, where the | operator allows us to express the fact that we are talking about two *different* subtrees with no need to exploit any notion of node identifier:

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

Referential integrity constraints can be exemplified by considering the following schema,

describing a list of books and a list of authors.

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

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.

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

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).

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, or that some constraints are not compatible with some schemas.

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 study on the manipulation of key constraint of (Buneman et al., 2001b), or the work about consistency between DTDs and constraints of (Fan and Libkin, 2001), in terms of the TQL logic. Of course, the real issue is the generalization of those result, to encompass a greater, or more natural, subset of TQL logic. To this aim, we plan to exploit the emerging results about algorithms for checking the validity of ambient logic formulas (Calcagno et al., 2003).

Although TQL is best suited to deal with constraints that are described in term of paths, we can also express and compute constraints that are defined at the type level, such as the UCM constraints defined in (Fan et al., 2001). In this case, however, a different syntax to describe mutually recursive formulas would be useful.

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 constitutes a high level specification of such an 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 a context valuation ρ, and returns the set of all the valuations ρ' such that $I \vDash_{\rho;\rho'} \mathcal{A}$. 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 subformulas of $\mathcal{A}$. We chose set-based evaluation because it is the only approach that guarantees reasonable performance in

presence of large amounts of data, hence is the forced choice for database 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 TQL implementation is described in (Conforti et al., 2002; Conforti et al., 2003), 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 that may be infinite. In the 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 (Kanellakis, 1995; Kuper et al., 2000). We are not going to describe it here, but more information can be found in (Conforti et al., 2002; Conforti et al., 2003). Moreover, the implementation directly supports the dualized logical operators indicated in the first table of Section 4.2. For the sake of simplicity, we assume here instead that all derived operators are rewritten in terms of the basic operators (which include, of course, negation) before execution.

Our query evaluation procedure shows how to directly evaluate a query to a resulting tree. 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. Then, 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 it is irrelevant or it 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.

Table 7.1. *The operators of table algebra*

$R^{\mathbf{V}} \cup^{\mathbf{V}} R'^{\mathbf{V}}$	$=_{def}\ R^{\mathbf{V}} \cup R'^{\mathbf{V}}$	$\subseteq \mathbf{1}^{\mathbf{V}}$
$Co^{\mathbf{V}}(R^{\mathbf{V}})$	$=_{def}\ \mathbf{1}^{\mathbf{V}} \setminus R^{\mathbf{V}}$	$\subseteq \mathbf{1}^{\mathbf{V}}$

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

Union, complement, product, and projection are completely standard. Since $\mathbf{1}^{\mathbf{V}}$ is infinite, the complement of a finite table is always infinite. The function $\rho^{\mathbf{V}}_+$, used to define restriction, denotes the function that coincides with $\rho^{\mathbf{V}}$ over $\mathbf{V}$, and maps every $\eta \notin \mathbf{V}$ to η. Hence, for example, $\sigma^{\mathbf{V}}_{x=n} R^{\mathbf{V}}$ returns all rows ρ in $R^{\mathbf{V}}$ such that $\rho(x) = n$. $\sigma^{\mathbf{V}}_{x=y} R^{\mathbf{V}}$ returns all rows in $R^{\mathbf{V}}$ such that $\rho(x) = \rho(y)$. $\sigma^{\mathbf{V}}_{n=n} R^{\mathbf{V}}$ returns $R^{\mathbf{V}}$, while $\sigma^{\mathbf{V}}_{n=m} R^{\mathbf{V}}$ returns $\emptyset$, if $n \neq m$.

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

Table 7.2. *Table algebra, derived operators*

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

Extension $Ext^{\mathbf{V}}_{\mathbf{V}'}(R^{\mathbf{V}})$ is a right-inverse of projection: it adds some new columns, and fills them with every possible value. Extending a table always produces an infinite table (unless $\mathbf{V} = \mathbf{V}'$). Intersection is standard, and is defined here by dualizing union. 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 use this operator, rather than extension plus intersection, in the definition of our query evaluation procedure.

7.2. *Query Evaluation*

We specify here the query evaluation procedure $\mathcal{Q}(Q)_\rho$ and the binder evaluation procedure $\mathcal{B}(I,\mathcal{A})_{\rho\gamma}$.

$\mathcal{B}(I,\mathcal{A})_{\rho\gamma}$ takes an information tree I and a formula $\mathcal{A}$ and yields a table R that contains all the valuations ρ' with schema $FV(\mathcal{A}) \setminus \mathbf{V}$ such that $I \vDash_{(\rho;\rho')} \mathcal{A}$.

$\mathcal{Q}(Q)_\rho$ takes a query Q and a row ρ that specifies a value for each free variable of Q, and 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. Then, the set of valuations $R = \mathcal{B}(I,\mathcal{A})_{\rho\epsilon}$ is computed. Finally, Q' is evaluated once for each row ρ of R; 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 binder evaluation procedure $\mathcal{B}(I,\mathcal{A})_{\rho\gamma}$ exploits a γ parameter to deal with recursive formulas. Since a formula specifies a function from trees to tables, γ maps each recursion variable ξ to a recursive function from information trees to tables. For any

value of γ, $\hat{\gamma}$ specifies its type, as follows:

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

γ is only used in the rules for ξ and $\mu\xi.\mathcal{A}$.

When $\mathcal{A}$ contains no free recursion variable, the schema of the table returned by $\mathcal{B}(I,\mathcal{A})_{\rho\mathbf{v},\gamma}$ is $FV(\mathcal{A}) \setminus \mathbf{V}$. The situation is more comples when $\mathcal{A}$ contains free recursion variables. For example, the schema of $\mathcal{B}(I,\xi)_{\rho\mathbf{v},\gamma}$ is $\hat{\gamma}(\xi)$. In general, the schema of the table returned by $\mathcal{B}(I,\mathcal{A})_{\rho\mathbf{v},\gamma}$ is given by $\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})$, where the schema function $\mathcal{S}$ is specified in Table 7.4, and enjoys the expected property that $\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\epsilon}) = FV(\mathcal{A}) \setminus \mathbf{V}$.

The notation $\{(x \mapsto n)\}$ represents a table that contains only the row $(x \mapsto n)$, and similarly for $\{(\mathcal{X} \mapsto I)\}$. Most definitions in Table 7.3 are easier to read if one ignores the schema information.

Table 7.3. *Binder and query evaluation*

$$
\begin{array}{lcll}
\mathcal{B}(I,\mathbf{0})_{\rho\mathbf{v},\gamma} & = & \textit{if } I = \mathbf{0} \textit{ then } \{\epsilon\} \textit{ else } \emptyset & \\
\mathcal{B}(I,n[\mathcal{A}])_{\rho\mathbf{v},\gamma} & = & \textit{if } I = n[I'] \textit{ then } \mathcal{B}(I',\mathcal{A})_{\rho\mathbf{v},\gamma} \textit{ 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} & = & \textit{if } I = n[I'] \textit{ then } \{(x \mapsto n)\} \bowtie^{\{x\},\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})} \mathcal{B}(I',\mathcal{A})_{\rho\mathbf{v},\gamma} \textit{else } \emptyset & \text{if } x \notin \mathbf{V} \\
\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{A} \mid \mathcal{B})_{\rho\mathbf{v},\gamma} & = & \bigcup^{\mathcal{S}(\mathcal{A}|\mathcal{B},\mathbf{V},\hat{\gamma})}_{I',I'' \in \{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{X})_{\rho\mathbf{v},\gamma} & = & \textit{if } I = \rho^{\mathbf{V}}(\mathcal{X}) \textit{ then } \{\epsilon\} \textit{ 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}
$$

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

Table 7.4. *The schema function* $\mathcal{S}$

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

We explain now the evaluation procedure.

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 empty 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 context 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 $[\![Q]\!]_{\rho^{\mathbf{V}}} \vDash_{\rho^{\mathbf{V}};\rho'^{\mathbf{V}'}} \mathcal{A}$ holds for any $\rho'^{\mathbf{V}'}$. Similarly, $\emptyset$ is the table that corresponds to the case when $[\![Q]\!]_{\rho^{\mathbf{V}}} \vDash_{\rho^{\mathbf{V}};\rho'^{\mathbf{V}'}} \mathcal{A}$ does *not* hold, for any $\rho'^{\mathbf{V}'}$.

The formula $n[\mathcal{A}]$ tests whether 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 context valuation. Otherwise, if I is not an edge, no binder is returned, as in the previous case. 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 $\sigma_{x=n} R'^{\mathbf{V}'}$.

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, in our implementation we actually minimize the use of this operator, by operating extensive query rewritings.

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 simple algorithm that tries to verify whether $\mathcal{A} \Rightarrow \exists x.\ x[\mathbf{T}]$; the simplicity and effectiveness of this algorithm is 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 context 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:

$$\rho \in \textstyle\prod^{\mathbf{V}}_{\mathbf{V}\setminus\{x\}} R^{\mathbf{V}} \ \Leftrightarrow\ \exists n \in \Lambda.\ (\rho; (x \mapsto n)) \in R^{\mathbf{V}}$$
$$\rho \in \textstyle\prod^{\mathbf{V}}_{\mathbf{V}\setminus\{\mathcal{X}\}} R^{\mathbf{V}} \ \Leftrightarrow\ \exists I \in \mathcal{IT}.\ (\rho; (\mathcal{X} \mapsto I)) \in R^{\mathbf{V}}$$

Since the rule for comparisons $\eta \sim \eta'$ is subtle, we expand some special cases in Table 7.5. The evaluation of $\eta \sim \eta'$ always 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).

Table 7.5. *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, we would express this as:

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

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 proved in detail in Appendix A.

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

Lemma 4. If $\mathcal{A}$ contains no free recursive variable, then:

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

Proof. The thesis follows immediately from the following property:

$$\begin{array}{l} dom(\Gamma) \text{ contains every free recursive variable in } \mathcal{A} \\ \wedge\ \forall \xi \in dom(\Gamma).\ \Gamma(\xi) = \emptyset \\ \Rightarrow\ \mathcal{S}(\mathcal{A}, \mathbf{V}, \Gamma) = FV(\mathcal{A}) \setminus \mathbf{V} \setminus dom(\Gamma) \end{array}$$

This property can be proved by induction on the definition of $\mathcal{S}(\mathcal{A}, \mathbf{V}, \Gamma)$ and by case inspection. □

Lemma 5.

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

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 6.

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

Proof. By induction on $\mathcal{A}$ and by cases. The only non-trivial case is recursion, where we resort to Lemma 5. □

We can now state the main lemma, which specifies the correctness of the binder evaluation procedure.

Lemma 7. 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, $\gamma(\xi_i) \in \mathcal{IT} \to \mathcal{T}^{\hat{\gamma}(\xi_i)}$, where $\hat{\gamma}(\xi_i)$ is disjoint from $\mathbf{V}$. then:

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

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

Proof. See Appendix A. □

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 $Q =$ *from* $Q' \vDash \mathcal{A}$ *select* Q''. By definition:

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

By induction, $\mathcal{Q}(Q')_{\rho\mathbf{v}} = [\![Q']\!]_{\rho\mathbf{v}}$. Hence, by Lemma 7, and using $\Xi = \emptyset$ ($\mathcal{A}$ contains no free recursion variables) we have:

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

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

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

The other cases are immediate.

□

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 labeled trees. Using the terminology of (Abiteboul et al., 1999), 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 mean that a formula $\mathcal{A}$ is always about one specific 'subject', which 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 describe just one tree and its structure, whereas it makes it more difficult to describe a relationship between two different trees. Many modal logics have been defined whose validity is decidable. This is not the case for TQL logic, but these logics may provide hints for the definition of a decidable but expressive sublogic.

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

— It makes it easy to say that two properties are satisfied by two disjoint subtrees, without using node or edge identity. For example, the following formula specifies that *title* is not a key: $\exists\mathcal{X}.\ .title[\mathcal{X}] \mid .title[\mathcal{X}]$.
— It makes it easy to describe record-like structures both partially ($b[] \mid c[] \mid \mathbf{T}$, meaning: contains $b[]$, $c[]$, and possibly more fields) and completely ($b[] \mid c[]$, meaning: 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': $paper[title[\mathbf{T}] \mid X]$.

This operator is what sets this logic apart from the other modal logics that have been proposed for querying semistructured data, or for reasoning about schemas and types for SSD, such as the logics proposed in (Alechina, 1999; Alechina et al., 2001; Calvanese et al., 2002). Another essential difference is the fact that modal logics are better suited to deal with graph structures, while our logic only deals with trees.

Composition is very similar to the $*$ operator of bunched logic and separation logic (O'Hearn and Pym, 1999; O'Hearn et al., 2001). These different logics have been defined independently of the ambient logic, with different motivations, but exhibit deep similarities. The most important technical difference between these logics and the one we presented here is that they only describe a flat horizontal structure, while the TQL logic adds a second dimension, using the $m[\mathcal{A}]$ operator, that allows one to describe a tree-shaped space.

TQL 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 (Fernandez et al., 1997; Fernandez et al., 1998), Lorel (Abiteboul et al., 1997; Goldman et al., 1999), XML-QL (Deutsch et al., 1999), Quilt (Chamberlin et al., 2000), XQuery (Boag et al., 2002) and, to some extent, YATL (Cluet et al., 1998). An in-depth comparison between the XML-QL YATL, and Lorel languages is carried out in (Fernandez et al., 1999), based on the analysis of thirteen typical queries. In (Ghelli, 2001) we write those same queries in TQL. For the thirteen queries in (Fernandez et al., 1999), their TQL description is quite similar to the corresponding XML-QL description, with a couple of exceptions. First, those XML-QL queries that in (Fernandez et al., 1999) 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. We express this query using the universal quantifier `foreach`: a pair `$X,$Y` matches the formula only if every author for `$X` is an author for `$Y` too. We separate the two `.book` formulas using | because, if we used $\wedge$, every book would appear in the result, paired with itself (the Lorel query that appears in (Fernandez et al., 1999) seems to exhibit this problem).

```
from   $Bib |= foreach $Z.  bib[ .book[$X And .author[$Z]]
                                |
                                .book[$Y And .author[$Z]]
                              ]
select pair[original[$X] | copy[$Y]]
```

The authors of (Fernandez et al., 1999) do not write the XML-QL version of this query, but they say that: "XML-QL can express this with a rather complex, nested query, which uses negation and the `isEmpty` predicate".

TQL is also better than XML-QL-like languages at expressing queries dealing with the non-existence of paths, such as 'find all the papers with no title' or 'find all papers whose only author, if any, is Buneman'. Lorel scores well in this case, thanks to the presence of universal quantification.

Quilt, XQuery, and XDuce (H. Hosoya, 2000) 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-XML-QL family.

The presence of universal quantification in the binding mechanism is not the most important difference between languages in this family and TQL. The most distinctive feature of TQL is the declarative, rather than procedural, nature of its binding mechanism.

As an example, consider the following query, which collects every work where Suciu plays a role, and inverts the name with the role:

```
bib[from $Bib |= .%*.$B[ $A[Suciu] | $Rest ]
    select $B[ Suciu[$A] | $Rest ]
   ]
```

In XQuery it would be expressed as follows,

```
<bib>
   for $b in $Bib//*,
     let $xx := $b/*,
       for $y in $xx
       where $y/data() = "Suciu"
   return <xf:name($b)>
                <Suciu>
                   xf:local-name($y)
                </Suciu>,
                { op:except($xx,$y)}
           </xf:name($b)>
<bib>
```

While the binding mechanism of XQuery requires the programmer to write a nested loop to specify how the bindings are extracted from the data, in TQL one only specifies the conditions that the variables *$B*, *$A*, and *$Rest*, should satisfy. This is even more evident if one considers the query that verifies whether *title* is a key:

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

In TQL, once the property is specified, the query is ready. In XQuery one has to do one

more step: a specific algorithm to check whether the property is true has to be designed, for example an algorithm that loops over all books, collects all titles in a multiset, and finally checks whether this multiset contains repeated elements. Moreover, as we have shown, the TQL program can be transformed in a program that *finds* every key, just by changing a constant into a variable. The XQuery program would have to be rewritten, since a different algorithm is needed.

Finally, a last essential 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).

Other logic-based languages for semistructured data have been defined with a precise semantics, but most of them focus on graphs rather than trees. For example, the language defined in (Bidoit and Ykhlef, 1999) is based on a first-order logic enriched with a fixpoint operator and with variables ranging over paths and graphs. The resulting language is very different from ours, both for its non-modal character and for being focused on graphs, although it may be interesting to try and import the idea of path variables in our logics.

9. Conclusions and Future Directions

We have defined a query language based on a logic that can also express types and constraints. Many important optimization problems are based on the analysis of the relationship between a query and a schema; a typical example is the removal of parts of a query that are incompatible with the schema. The use of the same logical language for queries, types, and constraints, should allow us to rephrase such problems in terms of implications or equivalences of logical formulas. Other schema and query analysis problems, like satisfiability of constraints (Arenas et al., 2002) or query path correctness (Colazzo et al., 2002), can be similarly expressed. We may then be able to use ambient logic techniques to approach these problems.

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 languages for semistructured data that enrich the regular-path-expression approach with mechanisms to describe tree shapes, instead of linear paths only. 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 (Cardelli, 1999). 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.

Acknowledgments

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. The implementation of TQL is mainly due to Francesco Pantaleo, Giovanni Conforti, Orlando Ferrara. Antonio Albano, Dario Colazzo, and Paolo Manghi, contributed to the TQL project in many ways.

References

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

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley, Reading, MA.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88.

Alechina, N. (1999). (Modal) logics for semistructed data. In Lambrix, P., Borgida, A., Lenzerini, M., Möller, R., and Patel-Schneider, P. F., editors, *Proc. of the Int. Workshop on Description Logics (DL'99), Linköping, Sweden*.

Alechina, N., Demri, S., and de Rijke, M. (2001). Path constraints from a modal logic point of view (extended abstract). In Lenzerini, M., Nardi, D., Nutt, W., and Suciu, D., editors, *Proc. of the 8th Int. Workshop on Knowledge Representation meets Databases (KRDB'01), Roma, Italy*.

Arenas, M., Fan, W., and Libkin, L. (2002). On verifying consistency of XML specifications. In *Proc. of the 21st Symposium on Principles of Database Systems (PODS)*, pages 259–270, New York. ACM Press.

Bidoit, N. and Ykhlef, M. (1999). Fixpoint calculus for querying semistructured data. In Atzeni, P., Mendelzon, A. O., and Mecca, G., editors, *Proc. of The World Wide Web and Databases (selected papers of the Int. Workshop WebDB'98)*, volume 1590 of *LNCS*, pages 78–97.

Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., Simon, J., and Stefanescu, M. (2002). XQuery 1.0: An XML query language. Available from http://www.w3c.org/TR/xquery.

Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., and Tan, W. C. (2001a). Keys for XML. In *Proc. of the 10th International World Wide Web Conference (WWW), Hong Kong, China*, pages 201–210.

Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., and Tan, W. C. (2001b). 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. Springer-Verlag.

Buneman, P., Davidson, S. B., Hillebrand, G. G., and Suciu, D. (1996). 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. *SIGMOD Record* 25(2), June 1996.

Buneman, P., Fan, W., Siméon, J., and Weinstein, S. (2001c). Constraints for semi-structured data and XML. *SIGMOD Record*, 30:47–54.

Caires, L. and Cardelli, L. (2003). A spatial logic for concurrency (part I). *Information and Computation*. To appear.

Calcagno, C., Cardelli, L., and Gordon, A. D. (2003). Deciding validity in a spatial logic for trees. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI), New Orleans, USA*.

Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2002). Description logics for information integration. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, number 2408 in LNCS, pages 41–60, Berlin. Springer-Verlag.

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

Cardelli, L., Gardner, P., and Ghelli, G. (2002). A spatial logic for querying graphs. In *Proc. of the 29th International Colloquium on Automata, Languages, and Programming (ICALP), Malaga, Spain*, number 2380 in LNCS, pages 597–610, Berlin. Springer-Verlag.

Cardelli, L., Gardner, P., and Ghelli, G. (2003). Manipulating trees with hidden labels. In *Proc. of Foundations of Software Science and Computation Structures (FOSSACS), Warsaw, Poland*.

Cardelli, L. and Gordon, A. D. (1998). Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag. Accepted for publication in *Theoretical Computer Science*.

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

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

Charatonik, W. and Talbot, J.-M. (2001). The decidability of model checking mobile ambients. In *Proc. of the 15th Annual Conference of the European Association for Computer Science Logic (EACSL)*, number 2142 in LNCS, pages 339–354, Berlin. Springer-Verlag.

Cluet, S., Delobel, C., Siméon, J., and Smaga, K. (1998). Your mediators need data conversion. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

Cohen, E. (2002). Validity and model checking for logics of finite multisets (draft). Unpublished note.

Colazzo, D., Ghelli, G., Manghi, P., and Sartiani, C. (2002). Types for correctness of queries over semistructured data. In *Proc. of the 5th Workshop on the Web and Data Bases (WebDB), Madison, Wisconsin, USA*, pages 13–18.

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

Conforti, G., Ferrara, O., and Ghelli, G. (2003). TQL algebra and its implementation (full paper). To appear.

Deutsch, A., M. Fernandez, D. F., Levy, A., and Suciu, D. (1999). A query language for XML. In *Proc. of the Eighth International World Wide Web Conference*.

Fan, W., Kuper, G. M., and Siméon, J. (2001). A unified constraint model for XML. In *World Wide Web*, pages 179–190.

Fan, W. and Libkin, L. (2001). On XML integrity constraints in the presence of DTDs. In ACM, editor, *Proc. of the 20th Symposium on Principles of Database Systems (PODS), Santa*

Barbara, California, SIGMOD Record, pages 114–125, New York, NY 10036, USA. ACM Press.

Fernandez, M., Florescu, D., Kang, J., Levy, A., and Suciu, D. (1998). 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.

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

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

Gelder, A. V. and Topor, R. W. (1991). Safety and translation of relational calculus queries. *ACM Transactions on Database Systems*, 16(2):235–278.

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

Goldman, R., McHugh, J., and Widom, J. (1999). 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.

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

Hosoya, H., Vouillon, J., and Pierce, B. C. (2000). Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22.

Kanellakis, P. (1995). 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.

Kuper, G., Libkin, L., and Paredaens, J. (2000). *Constraint Databases*. Springer-Verlag, Berlin.

O'Hearn, Reynolds, and Yang (2001). Local reasoning about programs that alter data structures. In *Proc. of the 15th Annual Conference of the European Association for Computer Science Logic (EACSL)*, number 2142 in LNCS, pages 1–, Berlin. Springer-Verlag.

O'Hearn, P. and Pym, D. (1999). The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244.

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

Ullman, J. D. (1982). *Principles of Database Systems, 2nd Edition*. Computer Science Press.

Ullman, J. D. (1988). *Principles of Database and Knowledge Base Systems, Volume I*. Computer Science Press.

W3C (2002a). XML Query use cases. http://www.w3.org/TR/xmlquery-use-cases.

W3C (2002b). XML Schema. http://www.w3.org/XML/Schema.

W3C (2002c). XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/TR/query-datamodel.

Appendix A. Proof of Lemma 7

Lemma. 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, $\gamma(\xi_i)\in\mathcal{IT}\to\mathcal{T}^{\hat\gamma(\xi_i)}$ 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 $\Leftrightarrow$.

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

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:

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

This is equivalent to:

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

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 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:

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

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

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

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

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

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

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

We now prove the opposite inclusion:

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

This is equivalent to:

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

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

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

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

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

and this, by induction, is equivalent to:

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

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

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

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

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

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

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

The two sides are equal. □

Appendix B. Table of equivalences

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 logical optimizer for queries.

Table B.1. *Some equations*

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