A Query Language Based on the Ambient Logic

Luca Cardelli[1] and Giorgio Ghelli[2]

[1] Microsoft Research, 1 Guildhall Street, Cambridge, UK
[2] Università di Pisa, Dipartimento di Informatica, Corso Italia 40, Pisa, Italy

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

1 Introduction

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

Unstructured collections, or unstructured data, are collections that do not respect a predefined schema, and hence need to carry a description of their own structure. These are called *semistructured* when one can recognize in them some degree of homogeneity. This partial regularity makes semistructured collections amenable to be accessed through query languages, but not through query languages that have been designed to access fully structured databases. New languages are needed that are able to tolerate the data irregularity, and that can be used to query, at the same time, both data and structure. Semistructured collections are usually modeled in terms of labeled graphs, or labeled trees [3].

The ambient logic is a modal logic proposed to describe the structural and computational properties of distributed and mobile computation [10]. 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. First order logic is a logic of predicates (i.e. relations) and therefore it is particularly suitable to describe relational data. But, to describe tree-shaped data, we need a more suitable logic: a logic of trees or graphs.

Here we define 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.

The paper is structured as follows. In this section we present a preview of the query language, and compare it with related proposals. In Section 2 we define the tree data model. In Section 3 we present the logic, upon which the query language, defined in Section 4, is defined. In Section 5 we present the evaluation model. In Section 6 we draw some conclusions.

1.1 A Preview

Consider the following bibliography, expressed in the syntax of our language TQL, which we explain in detail later. Informally, $a[F]$ represents a piece of data labeled a with contents F. The contents can be a collection of similar pieces of data, separated by "|". When the collection is empty, we can omit the brackets, so that, for example, $POPL[\,]$ can be written as $POPL$.

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

$$
\begin{array}{l}
ARTICLES = \\
\quad article[\ author[Cardelli] \mid author[Gordon] \mid title[Anytime\_Anywhere] \\
\qquad\qquad \mid conference[POPL] \mid year[2000] \\
\qquad\qquad \mid keyword[Ambient\_Calculus] \mid keyword[Logic]\ ] \mid \\
\quad article[\ author[Cardelli] \mid title[Wide\_Area\_Computation] \\
\qquad\qquad \mid booktitle[ICALP] \mid year[1999] \mid pages[403\text{-}444] \mid publisher[SV]\ ] \mid \\
\quad article[\ author[Ghelli] \mid author[Pierce] \mid title[Bounded\_Existentials] \\
\qquad\qquad \mid journal[TCS] \mid year[1998]\ ]
\end{array}
$$

Suppose we want to find all the papers in $ARTICLES$ where one author is *Cardelli*; then we can write the following query:

$$
\begin{array}{ll}
from & ARTICLES \vDash .article[X] \\
 & X \vDash .author[Cardelli] \\
select & paper[X]
\end{array}
$$

The query consists of a list of *matching expressions* contained between *from* and *select*, and a *reconstruction expression*, following *select*. The matching expressions bind X with every piece of data that is reachable from the root $ARTICLES$ through an *article* path, and such that a path *author* goes from X to *Cardelli*; the answer is $paper[author[Cardelli] \mid author[Gordon] \mid \ldots] \mid paper[author[Cardelli] \mid title[Wide\ Area\ Computation] \mid \ldots]$, i.e. the first two articles in the databases, with the outer *article* rewritten as *paper*.

This query language is characterized by the fact that a matching expression is actually a logic expression combining matching and logical operators. For example, the following query combines path expressions and logical implication ($\Rightarrow$) to retrieve papers with no other author then *Cardelli*. Informally, $\mathbf{T}$ matches anything, hence the second condition says: if X is an author, then it is *Cardelli*.

$$\begin{array}{ll} \textit{from} & \textit{ARTICLES} \vDash .\textit{article}[X] \\ & X \vDash .\textit{author}[\mathbf{T}] \Rightarrow .\textit{author}[\textit{Cardelli}] \\ \textit{select} & X \end{array}$$

Moreover, queries can be nested, giving us the power to restructure the collection, as we explain later.

1.2 Comparisons with Related Proposals

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

The tree logic can be compared with standard first order formalizations of labelled trees. Using the terminology of [3], 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 $a[]$" by saying:

$$\exists y.\ \textit{Ref}(x,a,y) \wedge (\forall y',y''.\ \neg \textit{Ref}(y,y',y'')) \wedge (\forall x',x''.\ x'' \neq y \Rightarrow \neg \textit{Ref}(x,x',x''))$$

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

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

Apart from modality, another feature of the ambient logic is that its fundamental operators deal with one-step paths ($a[\mathcal{A}]$) and with the composition of trees ($\mathcal{A} \mid \mathcal{A}'$), whereas the first order approach describes everything in terms of one-step paths ($\textit{Ref}(o1,a,o2)$). Composition is a powerful operator, at least for the following purposes:

- it makes it easy to describe record-like structures both partially ($b[] \mid c[] \mid \mathbf{T}$ means: contains $b[]$, $c[]$, and possibly more fields) and completely ($b[] \mid c[]$ means: contains $b[], c[]$ and only $b[], c[]$); complete descriptions are difficult in the path based approach;
- it makes it possible to bind a variable to 'the rest of the record', as in 'X is everything but the title': $\textit{paper}[\textit{title}[\mathbf{T}] \mid X]$.

The query language we described derives its essential *from-select* structure from set-theoretics comprehension, in the SQL tradition, and this makes it similar to other query languages for semistructured data, such as StruQL [14, 15], Lorel [5, 18], XML-QL [13], Quilt [11], and, to some extent, YATL [12]. An in-depth comparison between the XML-QL, YATL, and Lorel languages is carried out in [16], based on the analysis of thirteen typical queries. In [17] we wrote down those same queries in TQL; the result of this comparison is that, for the thirteen queries in [16], their TQL expression is very similar to the corresponding XML-QL, with a couple of exceptions. First, those XML-QL queries that, in [16], are expressed using Skolem functions, have to be expressed in a different way in TQL, since we do not have Skolem functions in the current version of TQL. However, our Skolem-free version of these queries is not complex. Second, XML-QL does not seem to have a general way of expressing universal quantification, and this problem shows up in the query that asks for pairs of books with the same set of authors; this is rather complex to express in XML-QL, but it is not difficult in TQL. Another related class of queries that are simpler to express using TQL are those related to the non-existence of paths, such as 'find all the papers with no title' or 'find all the papers whose only author, if any, is Ghelli'. Lorel does not have these problems, since it allows universal quantification. Quilt and XDuce [19] are Turing complete, hence are more expressive than the other languages we cited here.

One important feature of TQL is that it has a clean semantic interpretation, which pays off in several ways. First, the semantics should make it easier 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.

2 Information Trees

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.

2.1 Information Trees

We represent labeled trees as nested multisets; this corresponds, of course, to unordered trees. Ordered trees (e.g. XML data) could be represented as nested lists. This option would have an impact on the logic, where the symmetric $\mathcal{A} \mid \mathcal{B}$ operator could be replaced by an asymmetric one, $\mathcal{A};\mathcal{B}$. This change might actually simplfy some aspects of the logic, but in this paper we stick to the original notion of unordered trees from [10], which also matches some recent directions in XML [1].

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 $\mathcal{IT}$;
- if m is in Λ and I is in $\mathcal{IT}$ then the singleton multiset $\{\langle m, I\rangle\}$ is in $\mathcal{IT}$;
- $\mathcal{IT}$ is closed under multiset union $\biguplus_{j\in J} M(j)$, where J is an index set, and $M \in J \to \mathcal{IT}$.

2.2 Information Terms

We denote finite information trees by the following syntax of information term (info-terms), borrowed from the ambient calculus [9]. We define a function $[\![F]\!]$ mapping the info-term F to the denoted information tree. To this aim, we define three operators, $\mathbf{0}$, $m[\_]$ and $|$, on the domain of the information trees, which we use to interpret the corresponding operations on info-terms.

Info-terms and their information tree meaning

$F ::=$ info-term
- $\mathbf{0}$ denoting the empty multiset
- $m[F]$ denoting the multiset $\{\langle m, F\rangle\}$
- $F \mid F$ denoting multiset union

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

We use Π to denote the set of all terms generated by this grammar, also using parentheses for precedence. We often abbreviate $m[\mathbf{0}]$ as $m[]$, or as m. We assume that Λ includes the disjoint union of each basic data type of interest (integers, strings...), hence $5[\mathbf{0}]$, or 5, is a legitimate info-term. We assume that "$|$" associates to the right, i.e. $F \mid F' \mid F''$ is read $F \mid (F' \mid F'')$.

2.3 Congruence over Info-Terms

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

Congruence over info-terms

$F \equiv F$

$F' \equiv F \Rightarrow F \equiv F'$

$F \equiv F', F' \equiv F'' \Rightarrow F \equiv F''$

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

$F \equiv F' \Rightarrow F \mid F'' \equiv F' \mid F''$

$F \mid \mathbf{0} \equiv F$

$F \mid F' \equiv F' \mid F$
$(F \mid F') \mid F'' \equiv F \mid (F' \mid F'')$

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

2.4 Information Trees, OEM Trees, UnQL Trees

We can compare our information trees with two popular models for semistructured data: OEM data [24] and UnQL trees [6]. The first obvious difference is that OEM and UnQL models can be used to represent both trees and graphs, while here we focus only on trees. We are currently working on extending our model to include labeled graphs as well, but we prefer to focus on the simpler issue of trees, which is rich enough to warrant a separate study.

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

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

3 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 [10]. The ambient logic is particularly attractive because it is equipped with a large set of logical laws for tree-like structures, in particular logical equivalences, that can provide a foundation for query rewriting rules and query optimization.

We start here from a subset of the ambient logic as presented in [10], but we enrich it with information tree variables, label comparison, and recursion.

3.1 Formulas

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

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

Formulas:

$\eta ::=$	label expression
$\quad n$	label constant
$\quad x$	label variable
$\mathcal{A}, \mathcal{B} ::=$	formula
$\quad \mathbf{0}$	empty tree
$\quad \eta[\mathcal{A}]$	location
$\quad \mathcal{A} \mid \mathcal{B}$	composition
$\quad \mathbf{T}$	true
$\quad \neg\mathcal{A}$	negation
$\quad \mathcal{A} \wedge \mathcal{B}$	conjunction
$\quad \mathcal{X}$	tree variable
$\quad \exists x.\mathcal{A}$	quantification over label variables
$\quad \exists \mathcal{X}.\mathcal{A}$	quantification over tree variables
$\quad \eta \sim \eta'$	label comparison
$\quad \xi$	recursion variable
$\quad \mu\xi.\mathcal{A}$	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.

Formulas as sets of information trees

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

This style of semantics makes it easier to define the semantics of recursive formulas. Some consequences of the semantic definition are detailed shortly.

$[\![\mathbf{0}]\!]_{\rho,\,\delta}$ is the singleton $\{\mathbf{0}\}$. $[\![\eta[\mathcal{A}]]\!]_{\rho,\,\delta}$ contains the information tree $m[I]$, if $m = \rho(\eta)$ and I is in $[\![\mathcal{A}]\!]_{\rho,\,\delta}$. (We assume that ρ maps any label in Λ to itself, so that we can apply ρ to η even when η is not a variable.) For each I in $[\![\mathcal{A}]\!]_{\rho,\,\delta}$

and I' in $[\![\mathcal{B}]\!]_{\rho,\,\delta}$, $[\![\mathcal{A} \mid \mathcal{B}]\!]_{\rho,\,\delta}$ contains the information tree $I \mid I'$. $[\![\mathbf{T}]\!]_{\rho,\,\delta}$ is the set of all information trees (while its negation $\mathbf{F}$ denotes the empty set). $[\![\neg\mathcal{A}]\!]_{\rho,\,\delta}$ is the complement of $[\![\mathcal{A}]\!]_{\rho,\,\delta}$ with respect to the set of all information trees $\mathcal{IT}$. I is in $[\![\mathcal{A} \wedge \mathcal{B}]\!]_{\rho,\,\delta}$ if it is in $[\![\mathcal{A}]\!]_{\rho,\,\delta}$ and in $[\![\mathcal{B}]\!]_{\rho,\,\delta}$. I is in $[\![\exists x.\mathcal{A}]\!]_{\rho,\,\delta}$ if there exists some value n for x such that I is in $[\![\mathcal{A}]\!]_{\rho[x\mapsto n],\,\delta}$. Here $\rho[x \mapsto n]$ denotes the subtitution that maps x to n and otherwise coincides with ρ. $[\![\eta \sim \eta']\!]_{\rho,\,\delta}$ is the set $\mathcal{IT}$ if the comparison holds, else it is the empty set. $[\![\mu\xi.\mathcal{A}]\!]_{\rho,\,\delta}$ is the least fixpoint (with respect to set inclusion) of the monotonic function that maps any set of information trees S to $[\![\mathcal{A}]\!]_{\rho,\,\delta[\xi\mapsto S]}$.

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

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

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

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

Some properties of satisfaction

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

3.2 Some Derived Formulas

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

Derived formulas:

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

$F \vDash m[\Rightarrow \mathcal{A}]$ 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 an article where *Ghelli* is an author: $F \vDash \mathit{article}[\mathit{author}[\mathit{Ghelli}]|\mathbf{T}]$
- If F is an article, then *Ghelli* is an author: $F \vDash \mathit{article}[\Rightarrow \mathit{author}[\mathit{Ghelli}]|\mathbf{T}]$

$F \vDash \mathcal{A} \mathbin{||} \mathcal{B}$ 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 composition of F into F' and F'', such that F' satisfies $\mathit{article}[\mathcal{A}]$, and F'' satsfies $\mathbf{T}$; i.e., there is an article inside F that satisfies $\mathcal{A}$: $F \vDash \mathit{article}[\mathcal{A}] \mid \mathbf{T}$
- For every decomposition of F into F' and F'', either F' satisfies $\mathit{article}[\Rightarrow \mathcal{A}]$, or F'' satisfies $\mathbf{F}$; i.e., every article inside F satisfies $\mathcal{A}$: $F \vDash \mathit{article}[\Rightarrow \mathcal{A}] \mathbin{||} \mathbf{F}$

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

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

Some properties of satisfaction for derived formulas

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

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 could be exploited by a query logical optimizer.

Some equations

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

3.3 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 also a more compact and readable way of expressing common queries, like those outlined in the previous section.

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

$$ARTICLES \vDash article[\mathcal{X} \wedge (author[Cardelli] \mid \mathbf{T})] \mid \mathbf{T}$$

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

$$ARTICLES \vDash .article(\mathcal{X}).author[Cardelli]$$

Our path expressions support also the following features:

– Universally quantified paths: $\mathcal{X}$ is an article and *every* author of $\mathcal{X}$ is Cardelli.

$$ARTICLES \vDash .article(\mathcal{X})!author[Cardelli]$$

- Label negation: $\mathcal{X}$ is an article where *Ghelli* is the value of a field, but is not the author.

$$ARTICLES \vDash .article(\mathcal{X}).(\neg author)[Ghelli]$$

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

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

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

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

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

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

We now define the syntax of paths and its interpretation.

Path formulas:

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

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

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:

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

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

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

3.4 Tree Logic and Schemas

Path formulas explore the vertical structure of trees. Our logic can also express easily horizontal structure, as is common in schemas for semistructured data. (E.g. in XML DTDs, XDuce [19] and XMLSchema [1]. However, the present version of our logic deals directly only with unordered structures.)

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

$\mathbf{0}$	the empty tree
$\mathcal{A} \mid \mathcal{B}$	an $\mathcal{A}$ next to a $\mathcal{B}$
$\mathcal{A} \vee \mathcal{B}$	either an $\mathcal{A}$ or a $\mathcal{B}$
$n[\mathcal{A}]$	an edge n leading to an $\mathcal{A}$
$\mathcal{A}^* =_{def} \mu\xi.\ \mathbf{0} \vee (\mathcal{A} \mid \xi)$	a finite multiset of zero or more $\mathcal{A}$'s
$\mathcal{A}^+ =_{def} \mathcal{A} \mid \mathcal{A}^*$	a finite multiset of one or more $\mathcal{A}$'s
$A? =_{def} \mathbf{0} \vee \mathcal{A}$	optionally an $\mathcal{A}$

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

4 The Tree Query Language

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

4.1 The Query Language

A query language should feature the following functionalities:

- binding and selection: a mechanism to select values from the database and to bind them to variables;

– construction of the result: a mechanism to build a result starting from the bindings collected during the previous stage.

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

TQL queries:

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

We allow some tree functions f, chosen from a set Φ of functions of type $\mathcal{IT} \to \mathcal{IT}$, 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;
– *op(I)*, where *op* is a commutative, associative integer function with a neutral element; if all the pairs in I have a shape $n[I']$, where n is a natural number, then *op(I)* combines all the n's using the *op* operation obtaining the integer r, and returns $r[\mathbf{0}]$.

In practice, these functions would include user-defined functions written in an external programming language.

4.2 Query Semantics

The semantics of a query is defined in the following table. The interesting case is the one for *from* $Q \vDash \mathcal{A}$ *select* Q'. In this case, the subquery Q' is evaluated once for each valuation ρ' that extends the input valuation ρ and such that $[\![Q]\!]_\rho \in [\![\mathcal{A}]\!]_{\rho',\epsilon}$; all the resulting trees are then combined using the $\mid$ operator. The notation $\rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}}$ means that $\mathbf{V}' \supseteq \mathbf{V}$ and that $\rho'^{\mathbf{V}'}$ and $\rho^{\mathbf{V}}$ coincide over $\mathbf{V}$. For $F \in R^{\mathbf{V}} \to \mathcal{IT}$, we define $Par_{\rho^{\mathbf{V}} \in R^{\mathbf{V}}}\ F(\rho^{\mathbf{V}}) =_{def} \biguplus_{\rho^{\mathbf{V}} \in R^{\mathbf{V}}} F(\rho^{\mathbf{V}})$, where $\uplus$ is multiset union, namely the information tree operator that is used to interpret $\mid$.

Query semantics

$[\![\mathcal{X}]\!]_{\rho^{\mathbf{V}}}$	$= \rho^{\mathbf{V}}(\mathcal{X})$
$[\![\mathbf{0}]\!]_{\rho^{\mathbf{V}}}$	$= \mathbf{0}$
$[\![Q \mid Q']\!]_{\rho^{\mathbf{V}}}$	$= [\![Q]\!]_{\rho^{\mathbf{V}}} \mid [\![Q']\!]_{\rho^{\mathbf{V}}}$

$$\llbracket m[Q] \rrbracket_{\rho^{\mathbf{V}}} = m[\llbracket Q \rrbracket_{\rho^{\mathbf{V}}}]$$
$$\llbracket x[Q] \rrbracket_{\rho^{\mathbf{V}}} = \rho^{\mathbf{V}}(x)[\llbracket Q \rrbracket_{\rho^{\mathbf{V}}}]$$
$$\llbracket f(Q) \rrbracket_{\rho^{\mathbf{V}}} = f(\llbracket Q \rrbracket_{\rho^{\mathbf{V}}})$$
$$\llbracket \mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ Q' \rrbracket_{\rho^{\mathbf{V}}}$$
$$= \ \mathit{Par}_{\rho'^{\mathbf{V}'} \in \{\rho'^{\mathbf{V}'} \ \mid\ \mathbf{V}'=\mathbf{V} \cup FV(\mathcal{A}),\ \rho'^{\mathbf{V}'} \supseteq \rho^{\mathbf{V}},\ \llbracket Q \rrbracket_{\rho^{\mathbf{V}}} \in \llbracket \mathcal{A} \rrbracket_{\rho'^{\mathbf{V}'},\epsilon}\}} \llbracket Q' \rrbracket_{\rho'^{\mathbf{V}'}}$$

According to this semantics, the result of a query *from* $Q' \vDash \mathcal{A}$ *select* Q'' can be an infinite multiset. Therefore, in a nested query, the database Q' can be infinite, even if we start from a finite initial database. Obviously, one would not like this to happen in practice. One possible solution is to syntactically restrict Q' to a variable $\mathcal{X}$. Another solution is to have a static or dynamic check on the finiteness of the result; one such option is dicussed in Section 4.4.

4.3 Examples of Queries

We explain the query operators through examples. As in Section 1.1, we abbreviate a query

$$\mathit{from}\ Q \vDash \mathcal{A}\ \mathit{select}\ \mathit{from}\ Q' \vDash \mathcal{A}'\ \mathit{select}\ Q''$$

as

$$\mathit{from}\ Q \vDash \mathcal{A},\ Q' \vDash \mathcal{A}'\ \mathit{select}\ Q''\ .$$

The database *ARTICLES* is the one given in Section 1.1.

All papers whose only author (if any) is *Cardelli* can be retrieved by the following query (where we use $\mathcal{X} \wedge \ldots$ as an alternative to a nested binder $\mathcal{X} \vDash \ldots$):

$$\mathit{from}\quad \mathit{ARTICLES} \vDash\ .\mathit{article}[\mathcal{X} \wedge\ !\mathit{author}[\mathit{Cardelli}]] \quad \mathit{select}\quad \mathcal{X}$$

We may use disjunction to find both *e-mail*s and *email*s inside some *author* field.

$$\begin{array}{ll} \mathit{from} & \mathit{ARTICLES} \vDash\ .\mathit{article}[.\mathit{author}[.\mathit{e\text{-}mail}[\mathcal{X}] \vee .\mathit{email}[\mathcal{X}]]] \\ \mathit{select} & \mathit{e\text{-}mail}[\mathcal{X}] \end{array}$$

Using recursion, we look for *e-mail* at the current level or, recursively, at any inner nesting level.[1]

$$\begin{array}{ll} \mathit{from} & \mathit{ARTICLES} \vDash\ \mu\xi.\ .\mathit{e\text{-}mail}[\mathcal{X}] \vee .\mathit{email}[\mathcal{X}] \vee \exists x.\ .x[\xi] \\ \mathit{select} & \mathit{e\text{-}mail}[\mathcal{X}] \end{array}$$

The following query binds two label variables y and z to the label and the content of a field $y[z]$, where z is '*like %Ghelli%*' (*like* matches '%' to any substring). Recursion may be used to look for such fields at any depth.

[1] When every $\mathcal{X}$ is inside an $m[]$ operator, like in this example, recursion is guaranteed to terminate, but we still have enough flexibility to express complex queries, such as queries that evaluate boolean circuits [22].

$$
\begin{array}{ll}
\mathit{from} & \mathit{ARTICLES} \vDash \ .\mathit{article}[.y[z] \wedge z \ \mathit{like} \ \%\mathit{Ghelli}\%] \\
\mathit{select} & \mathit{found}[\mathit{label}[y] \mid \mathit{content}[z]]
\end{array}
$$

Query nesting allows us to restructure data. For example, the following query rearranges papers according to their year of publication: for each year $\mathcal{X}$ (outer *from*), it collects all the papers of that year. The composition $\mathit{Year}[\mathcal{X}] \mid \mathcal{Z}$ binds $\mathcal{Z}$ to all fields but the year; this way of collecting all the siblings except one is impossible, or difficult, in most other query languages.

$$
\begin{array}{ll}
\mathit{from} & \mathit{ARTICLES} \vDash .\mathit{article}[.\mathit{Year}[\mathcal{X}]] \\
\mathit{select} & \mathit{publications\_by\_year}[\ \mathit{Year}[\mathcal{X}] \\
& \qquad \mid \ (\mathit{from} \ \ \mathit{ARTICLES} \vDash .\mathit{article}[\mathit{Year}[\mathcal{X}] \ \mid \mathcal{Z}] \\
& \qquad \quad \ \mathit{select} \ \ \mathit{article}[\mathcal{Z}] \ ) \\
& \quad]
\end{array}
$$

Relational-style join queries can be easily written in TQL either by matching the two data sources with two logical expressions that share some variables (equi-joins) or by exploiting the comparison operators. Universal quantification can be expressed both on label and tree variables; more examples can be found in [17].

4.4 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:

$$\mathit{from} \ \ db \vDash (\mathit{author}[\mathcal{X}] \vee \mathit{autore}[\mathcal{Y}]) \mid \mathbf{T} \ \ \ \mathit{select} \ \ \ \mathit{author}[\mathcal{X}] \mid \mathit{autore}[\mathcal{Y}]$$

Intuitively, every entry in *db* that is an *author* binds $\mathcal{X}$ but not $\mathcal{Y}$, and vice-versa for *autore* entries. Formally, both situations generate an infinite amount of valuations; for example, if $\rho(db) = \mathit{author}[m[]]$, then $\{\rho' \mid [\![db]\!]_\rho \in [\![\mathcal{A}]\!]_{\rho',\epsilon}\}$ is the infinite set

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

Negation creates a similar problem. Consider the following query.

$$\mathit{from} \ \ db \vDash \neg \mathit{author}[\mathcal{X}] \ \ \ \mathit{select} \ \ \ \mathit{notauthor}[\mathcal{X}]$$

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

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

and the query has the following infinite result:

$$\{\mathit{notauthor}[I] \ \mid \ I \in \mathcal{IT}, I \neq m[]\} \ .$$

These queries present two different, but related, problems:

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

We say that a query is safe when its semantics is finite. Query safety is known to be undecidable for the relational tuple calculus [4], and we suspect it is undecidable for our calculus too. However, as in relational calculi, it is not difficult to devise some sufficient syntactical conditions for safety, and to solve the non-safety problem by restricting the language to the syntactically safe queries. A different way to solve the problem is to allow unsafe queries, and to design a query processor for them. Our semantics accounts for unsafe queries, since it does not restrict the set of valuations generated by a binder to be finite, nor does it restrict the query answer to be finite.

5 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 4.2 and an implementation algorithm. It is based on an algebra of trees and tables that is suggestive of realistic implementations, and may be seen as a specification of such implementations. In Pisa we have realized one such implementation, which is described in [23, 8].

The query evaluation procedure is based on the manipulation of sets of valuations. These sets, unfortunately, may be infinite. For a real implementation, one must typically find a finite representation of infinite sets. Moreover, at the level of query manipulations, one would like to push negation to the leaves, introducing dualized logical operators as indicated in the first table in Section 3.2. These dualized operators also become part of an implementation. We do not deal here with the possible ways of finitely representing these infinite sets, or how to implement operators over them. In [23, 8], though, we describe a technique for finitely representing sets of valuations in terms of a finite disjunction of a set of conjunctive constraints over the valuations, in the style of [20, 21].

Any practical implementation of a query language is based on the use of particular efficiently implementable operators, such as relational join and union. We write our query evaluation procedure in this style as much as possible, but we naively use set complement to interpret negation, and we do not deal with dualized operators.

Our query evaluation procedure shows how to directly evaluate a query to a resulting set of trees. In database technology, instead, it is typical to translate the query into an expression over algebraic operators (which, in [23, 8] and in XML Query Algebra [2], include also operators such as if-then-else, iteration and fixpoint). These expressions are first syntactically manipulated to enhance their performance, and finally evaluated. We ignore here issues of translation and manipulation of intermediate representations.

The core of the query evaluation problem is binder evaluation. A binder evaluation procedure takes an information tree I and a formula $\mathcal{A}$, that is used as a pattern for matching against I. The procedure takes also a valuation ρ and returns the set of all the valuations for the free variables of $\mathcal{A}$ that are not in the domain of ρ.

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.

5.1 The Table Algebra

Let $\mathbf{V} = V_1, ..., V_n$ be a finite set of variables, where each variable 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.

A row with schema $\mathbf{V}$ is a function that maps each V_i to an element of $U(V_i)$; we use $\rho^{\mathbf{V}}$ as a meta-variable to range over rows with schema $\mathbf{V}$ (or just ρ when $\mathbf{V}$ is clear from context). A table with schema $\mathbf{V}$ is a set of rows over $\mathbf{V}$; we use $\mathcal{T}^{\mathbf{V}}$ for the set of tables with schema $\mathbf{V}$, and $R^{\mathbf{V}}$ as a meta-variable to range over $\mathcal{T}^{\mathbf{V}}$. 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, $\{\epsilon\}$. We use $\mathbf{1}^{\mathbf{V}}$ to denote the largest table with schema $\mathbf{V}$, i.e. the set of all rows with schema $\mathbf{V}$.

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

The operators of table algebra:

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

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

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

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

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

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

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

where $\sim$ is a label comparison operator, as in Section 3.

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

Table algebra, derived operators:

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

The operator $R^{\mathbf{V}} \ \bowtie^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$ is well-known in the database field. It is called 'natural join', and can be also defined as follows: the set containing all rows obtained by concatenating each row ρ in $R^{\mathbf{V}}$ with those rows ρ' in $R'^{\mathbf{V}'}$ such that ρ and ρ' coincide over $\mathbf{V} \cap \mathbf{V}'$. One important property of natural join is that it always yields finite tables when is applied to finite tables, even if its definition uses the extension operator. Moreover, the optimization of join has been extensively studied; for this reason we will use this operator, rather than extension plus intersection, in the definition of our query evaluation procedure.

Outer union $R^{\mathbf{V}} \oplus^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'}$ and co-projection $\coprod^{\mathbf{V}}_{\mathbf{V}'} R^{\mathbf{V}}$ are useful for treating the dualized operators.

Outer union is dual to join, in the following sense:

$$R^{\mathbf{V}} \ \oplus^{\mathbf{V},\mathbf{V}'} R'^{\mathbf{V}'} \ = \ \mathit{Co}^{\mathbf{V}\cup\mathbf{V}'}(\mathit{Co}^{\mathbf{V}}(R^{\mathbf{V}}) \ \bowtie^{\mathbf{V},\mathbf{V}'} \mathit{Co}^{\mathbf{V}'}(R'^{\mathbf{V}'}))$$

Projection and co-projection are both left-inverse of extension:

$$\prod^{\mathbf{V}'}_{\mathbf{V}}(\mathit{Ext}^{\mathbf{V}}_{\mathbf{V}'}(R^{\mathbf{V}})) = R^{\mathbf{V}}$$
$$\coprod^{\mathbf{V}'}_{\mathbf{V}}(\mathit{Ext}^{\mathbf{V}}_{\mathbf{V}'}(R^{\mathbf{V}})) = R^{\mathbf{V}}$$

However, they represent two different ways of right-inverting extension:

$$\prod^{\mathbf{V}}_{\mathbf{V}'} R^{\mathbf{V}} = \bigcap\{R'^{\mathbf{V}'} \mid \mathit{Ext}^{\mathbf{V}'}_{\mathbf{V}}(R'^{\mathbf{V}'}) \supseteq R^{\mathbf{V}}\}$$
$$\coprod^{\mathbf{V}}_{\mathbf{V}'} R^{\mathbf{V}} = \bigcup\{R'^{\mathbf{V}'} \mid \mathit{Ext}^{\mathbf{V}'}_{\mathbf{V}}(R'^{\mathbf{V}'}) \subseteq R^{\mathbf{V}}\}$$

5.2 Query Evaluation

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

The first part of the table describes how a quadruple $I, \mathcal{A}, \rho^{\mathbf{V}}, \gamma$ is evaluated by a binder evaluation procedure $\mathcal{B}$ to return a table with schema $\mathcal{S}(\mathcal{A}, \mathbf{V}, \hat{\gamma})$. The schema function $\mathcal{S}$ is specified in the table that follows, and enjoys the property that $\mathcal{S}(\mathcal{A}, \mathbf{V}, \hat{\epsilon}) = FV(\mathcal{A}) \setminus \mathbf{V}$. Here γ is an environment that maps recursion variables ξ to functions from information trees to tables. We assume that γ is always given together with a schema $\hat{\gamma}$ mapping recursion variables to sets of variables $\mathbf{V}$, such that $\gamma(\xi) \in \mathcal{IT} \to \mathcal{T}^{\hat{\gamma}(\xi)}$.

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

Binder and query evaluation

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

$$
\begin{array}{ll}
\mathcal{Q}(\mathcal{X})_{\rho^{\mathbf{V}}} & = \rho^{\mathbf{V}}(\mathcal{X}) \\
\mathcal{Q}(\mathbf{0})_{\rho^{\mathbf{V}}} & = \mathbf{0} \\
\mathcal{Q}(Q \mid Q')_{\rho^{\mathbf{V}}} & = \mathcal{Q}(Q)_{\rho^{\mathbf{V}}} \mid \mathcal{Q}(Q')_{\rho^{\mathbf{V}}} \\
\mathcal{Q}(m[Q])_{\rho^{\mathbf{V}}} & = m[\mathcal{Q}(Q)_{\rho^{\mathbf{V}}}] \\
\mathcal{Q}(x[Q])_{\rho^{\mathbf{V}}} & = \rho^{\mathbf{V}}(x)[\mathcal{Q}(Q)_{\rho^{\mathbf{V}}}] \\
\mathcal{Q}(f(Q))_{\rho^{\mathbf{V}}} & = f(\mathcal{Q}(Q)_{\rho^{\mathbf{V}}})
\end{array}
$$

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

The schema function $\mathcal{S}$

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

Since the rule for comparisons $\eta \sim \eta'$ is subtle, we expand here some special cases.

Some special cases of comparison evaluation

$$\begin{array}{lll}
\mathcal{B}(I, x \sim x')_{\rho^{\mathbf{V}},\gamma} = & \sigma^{\{x,x'\}}_{x \sim x'} \mathbf{1}^{\{x,x'\}} & \text{if } x \notin \mathbf{V}, x' \notin \mathbf{V} \\
\mathcal{B}(I, x \sim x')_{\rho^{\mathbf{V}},\gamma} = & \sigma^{\{x\}}_{x \sim \rho^{\mathbf{V}}(x')} \mathbf{1}^{\{x\}} & \text{if } x \notin \mathbf{V}, x' \in \mathbf{V} \\
\mathcal{B}(I, x \sim x')_{\rho^{\mathbf{V}},\gamma} = & \sigma^{\emptyset}_{\rho^{\mathbf{V}}(x) \sim \rho^{\mathbf{V}}(x')} \mathbf{1}^{\emptyset} & \text{if } x \in \mathbf{V}, x' \in \mathbf{V} \\
\mathcal{B}(I, x \sim n)_{\rho^{\mathbf{V}},\gamma} = & \sigma^{\{x\}}_{x \sim n} \mathbf{1}^{\{x\}} & \text{if } x \notin \mathbf{V} \\
\mathcal{B}(I, n \sim n')_{\rho^{\mathbf{V}},\gamma} = & \sigma^{\emptyset}_{n \sim n'} \mathbf{1}^{\emptyset} & (\textit{i.e. if } n \sim n' \textit{ then } \{\epsilon\} \textit{ else } \emptyset)
\end{array}$$

Lemma 1.

$$\mathcal{S}(\mu\xi.\mathcal{A},\mathbf{V},\hat{\gamma}) = \mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma}[\xi \mapsto \mathcal{S}(\mu\xi.\mathcal{A},\mathbf{V},\hat{\gamma})])$$
$$\mathcal{B}(I,\mathcal{A})_{\rho^{\mathbf{V}},\gamma} \in \mathcal{T}^{\mathcal{S}(\mathcal{A},\mathbf{V},\hat{\gamma})}.$$

Lemma 2. *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:*

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

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

The following proposition states that the query evaluation procedure is equivalent to the query semantics of Section 4.2. The proof uses Lemma 2 in the from-select case.

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

6 Conclusions and Future Directions

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

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

In this study we have exploited a subset of the ambient logic. The ambient logic, and the calculus, also offer operators to specify and perform tree updates [7]. Possible connections with semistructured data updates should be explored.

An implementation of TQL is currently being carried out, based on the implementation model we described. The current prototype can be used to query XML documents accessible through files or through web servers.

Acknowledgements Andrew D. Gordon contributed to this work with many useful suggestions. Giorgio Ghelli was partially supported by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica", project DATA-X, by Microsoft Research, and by the E.U. workgroup APPSEM.

References

1. XML schema. Available from http://www.w3c.org, 2000.
2. XML query. Available from http://www.w3c.org, 2001.
3. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the WEB: From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Mateo, CA, October 1999.
4. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.
5. Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
6. P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD), Montreal, Quebec, Canada*, pages 505–516, 4–6 June 1996. *SIGMOD Record* 25(2), June 1996.

7. L. Cardelli. Semistructured computation. In *Proc. of the Seventh Intl. Workshop on Data Base Programming Languages (DBPL)*, 1999.
8. L. Cardelli and G. Ghelli. Evaluation of TQL queries. Available from http://www.di.unipi.it/~ghelli/papers.html, 2001.
9. L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998. Accepted for publication in *Theoretical Computer Science*.
10. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. of Principles of Programming Languages (POPL)*. ACM Press, January 2000.
11. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proc. of Workshop on the Web and Data Bases (WebDB)*, 2000.
12. Sophie Cluet, Claude Delobel, Jrme Simon, and Katarzyna Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1998.
13. A. Deutsch, D. Florescu M. Fernandez, A. Levy, and D. Suciu. A query language for XML. In *Proc. of the Eighth International World Wide Web Conference*, 1999.
14. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a web-site management system. In *Proc. of Workshop on Management of Semistructured Data, Tucson*, 1997.
15. Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 414–425, 1998.
16. Mary Fernandez, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages: Experiences and exemplars. Available from http://www-db.research.bell-labs.com/user/simeon/xquery.ps, 1999.
17. G. Ghelli. TQL as an XML query language. Available from http://www.di.unipi.it/~ghelli/papers.html, 2001.
18. R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Proc. of Workshop on the Web and Data Bases (WebDB)*, pages 25–30, 1999.
19. B.C. Pierce H. Hosoya. XDuce: A typed XML processing language (preliminary report). In *Proc. of Workshop on the Web and Data Bases (WebDB)*, 2000.
20. P. Kanellakis. Tutorial: Constraint programming and database languages. In *Proc. of the 14th Symposium on Principles of Database Systems (PODS), San Jose, California*, pages 46–53. ACM Press, 1995.
21. G. Kuper, L. Libkin, and J. Paredaens. *Constraint Databases*. Springer-Verlag, Berlin, 2000.
22. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. of the 19th Symposium on Principles of Database Systems (PODS)*, 2000.
23. F. Pantaleo. Realizzazione di un linguaggio di interrogazione per XML. Tesi di Laurea del Dipartimento di Informatica dell'Università di Pisa, 2000.
24. Y. Papakonstantinou, H.G. Molina, and J. Widom. Object exchange across heterogeneous information sources. *Proc. of the eleventh IEEE Int. Conference on Data Engineering, Birmingham, England*, pages 251–260, 1996.