A Query Language for Semistructured Data Based on the Ambient Logic

Luca Cardelli
Microsoft Research

Giorgio Ghelli
Università di Pisa

Preliminary version, April 4, 2000

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 an ideal 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 describe an execution model for this language. The language turns out to be quite expressive. Its strong foundation, and the wealth of results that have been proved about 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. We call them *semi*structured when, within those collections, one can recognize pieces of data whose structure presents 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 irregularity in the data, and that can be used to query both the data and its structure. Semistructured collections are usually modeled in terms of labeled graphs, or labeled trees [ABS99].

The ambient logic is a modal logic proposed to describe the structural and computational properties of distributed and mobile computation [CG00]. The logic comes equipped with a rich set of results. In particular, the issues of logical implication and equivalence have been studied, and inference systems have been devised. The structural part of the ambient logic is, essentially, a logic designed to describe properties of labeled trees. It is therefore an ideal foundation for query languages for semistructured data, much in the same way as first order logic is a fitting foundation for relational query languages.

Here we define a query language for semistructured data that is based on the ambient logic, and we describe an execution model for this language. The language turns out to be quite expressive. Its strong foundation, and the wealth of results that have already been produced in relation to 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. 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 present a subset of *restricted queries* which enjoy a *safety* property. In Section 7 we hint at some extensions needed to deal with XML data. In Section 8 we compare our language with related proposals. In Section 9 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 $POPL[\,]$ 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.

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

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

$from \quad ARTICLES \vDash .article[X]$
$\qquad\quad X \vDash .author[Cardelli]$
$select \quad paper[X]$

The query consists of a list of *matching expressions* $ARTICLES \vDash .article[X]$, $X \vDash .author[Cardelli]$ 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 $Cardelli$.

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 to retrieve papers with no other author then $Cardelli$. Informally, $\mathbf{T}$ matches everything, hence the second condition says: if X is an author, then it is $Cardelli$.

$from \quad ARTICLES \vDash .article[X]$
$\qquad\quad X \vDash .author[\mathbf{T}] => .author[Cardelli]$
$select \quad X$

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

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 we define an equivalence relation that deteremines when two different expressions denote the same information tree.

2.1 Information trees

We represent trees as nested multisets. For a given set of *labels* Λ, we define the set of information trees as the smallest collection $\mathcal{IT}$ such that:

- the empty multiset, $\{\}$, is in $\mathcal{IT}$;
- if m is in Λ and F is in $\mathcal{IT}$ then the singleton multiset $\{\langle m, F\rangle\}$ is in $\mathcal{IT}$;
- $\mathcal{IT}$ is closed under binary multiset union.

2.2 Info-terms

We denote information trees by the following syntax of information term (info-terms), borrowed from the ambient calculus [CG98].

Info-terms and their information tree meaning

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

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 $F \mid (F' \mid F'')$.

We can visualize an information tree either as an unordered edge-labeled tree, or as a forest of node-labeled trees. In this paper we adopt the first view, hence we say that $b[\mathbf{0}] \mid c[\mathbf{0}]$ is a tree with two edges b and c going from the root to two leaves.

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

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

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

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 [PMW96] and UnQL trees [BDHS96]. 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 describing spatial and temporal properties of the mobile processes that can be specified through the ambient calculus [CG00]. The ambient logic is particularly attractive because it is equipped with a large set of logical laws, 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 [CG00], 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 ξ to its binder.

Formulas:

$\eta ::=$	label expression
$\quad n$	$\quad$ label constant
$\quad x$	$\quad$ label variable
$\mathcal{A}, \mathcal{B} ::=$	formula
$\quad \mathbf{0}$	$\quad$ empty tree
$\quad \eta[\mathcal{A}]$	$\quad$ location
$\quad \mathcal{A} \mid \mathcal{B}$	$\quad$ composition

$\mathbf{T}$	true
$\neg\mathcal{A}$	negation
$\mathcal{A} \wedge \mathcal{B}$	conjunction
$\mathcal{X}$	tree variable
$\exists x.\mathcal{A}$	quantification over labels
$\exists \mathcal{X}.\mathcal{A}$	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 $[\![\mathcal{A}]\!]_\sigma$, which maps $\mathcal{A}$ to a set of information trees with respect to a substitution σ over the free variables of $\mathcal{A}$. The substitution σ maps label variables x to labels (elements of Λ), tree variables $\mathcal{X}$ to information trees, and recursion variables ξ to sets of information trees. This style of semantics makes it easier to define the semantics of recursive formulas. Some consequences of the semantic definition are detailed shortly.

Formulas as sets of information trees

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

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

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

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

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

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

Some properties of satisfaction

$$\begin{array}{lll}
F \vDash_\sigma \mathbf{0} & \Leftrightarrow & F \equiv \mathbf{0} \\
F \vDash_\sigma \eta[\mathcal{A}] & \Leftrightarrow & \exists F'.\ F \equiv \sigma(\eta)[F'] \ \wedge\ F' \vDash_\sigma \mathcal{A} \\
F \vDash_\sigma \mathcal{A} \mid \mathcal{B} & \Leftrightarrow & \exists F', F''.\ F \equiv F' \mid F'' \ \wedge\ F' \vDash_\sigma \mathcal{A} \ \wedge\ F'' \vDash_\sigma \mathcal{B} \\
F \vDash_\sigma \mathbf{T} & & \\
F \vDash_\sigma \neg\mathcal{A} & \Leftrightarrow & \neg F \vDash_\sigma \mathcal{A} \\
F \vDash_\sigma \mathcal{A} \wedge \mathcal{B} & \Leftrightarrow & F \vDash_\sigma \mathcal{A} \ \wedge\ F \vDash_\sigma \mathcal{B} \\
F \vDash_\sigma \exists x.\mathcal{A} & \Leftrightarrow & \exists m \in \Lambda.\ F \vDash_\sigma \mathcal{A}\{x \leftarrow m\} \\
F \vDash_\sigma \exists \mathcal{X}.\mathcal{A} & \Leftrightarrow & \exists F \in \mathcal{IT}.\ F \vDash_\sigma \mathcal{A}\{\mathcal{X} \leftarrow F\} \\
F \vDash_\sigma \eta \sim \eta' & \Leftrightarrow & \sigma(\eta) \sim \sigma(\eta') \\
F \vDash_\sigma \mu\xi.\mathcal{A} & \Leftrightarrow & F \vDash_\sigma \mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\} \\
F \vDash_\sigma \mathcal{X} & \Leftrightarrow & F \equiv \sigma(\mathcal{X})
\end{array}$$

3.2 Some derived formulas

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

Derived formulas:

$$\begin{array}{lll@{\qquad\qquad}lll}
n[\Rightarrow \mathcal{A}] & =_{def} & \neg(n[\neg\mathcal{A}]) & \mathcal{A} \mid\mid \mathcal{B} & =_{def} & \neg(\neg\mathcal{A} \mid \neg\mathcal{B}) \\
\mathbf{F} & =_{def} & \neg\mathbf{T} & \mathcal{A} \vee \mathcal{B} & =_{def} & \neg(\neg\mathcal{A} \wedge \neg\mathcal{B}) \\
\forall x.\mathcal{A} & =_{def} & \neg(\exists x.\neg\mathcal{A}) & \forall \mathcal{X}.\mathcal{A} & =_{def} & \neg(\exists \mathcal{X}.\neg\mathcal{A}) \\
\nu\xi.\mathcal{A} & =_{def} & \neg(\mu\xi.\neg\mathcal{A}\{\xi \leftarrow \neg\xi\}) & & &
\end{array}$$

$F \vDash m[\Rightarrow \mathcal{A}]$ 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: $F \vDash \mathit{article}[\Rightarrow \mathit{author}[\mathit{Ghelli}]|\mathbf{T}]$

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

- There exists a decomposition of F into F' and F'', such that F' satisfies $\mathit{article}[\mathcal{A}]$, 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 $article[\Rightarrow \mathcal{A}]$, or F'' satisfies $\mathbf{F}$; i.e., every article inside F satisfies $\mathcal{A}$: $F \vDash article[\Rightarrow \mathcal{A}] \mid\mid \mathbf{F}$

The dual of the least fixpoint operator $\mu\xi.\mathcal{A}$ is the greatest fixpoint operator $\nu\xi.\mathcal{A}$. For example $\mu\xi.\xi$ is equivalent to $\mathbf{F}$, while $\nu\xi.\xi$ is equivalent to $\mathbf{T}$. More interestingly, $\mu\xi.\mathbf{0} \vee m[\xi]$ describes every information tree that matches $m[m[\ldots m[]]]$, and, on finite trees, it is equivalent to $\nu\xi.\mathbf{0} \vee m[\xi]$. However, if we consider infinite trees, the distinction between least and greatest fixpoint becomes more important. For example, the infinite tree $m[m[\ldots]]$ satisfies $\nu\xi.\mathbf{0} \vee m[\xi]$, but does not satisfy $\mu\xi.\mathbf{0} \vee m[\xi]$. When we consider only finite trees, 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 form the basis for a matching algorithm.

Some properties of satisfaction for derived formulas

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

Many equations have been proved for the ambient logic, and are inherited by the tree logic. We list here some of them. Such equations would be the basic tool for a logical optimizer.

Some equations

$$
\begin{array}{lll@{\qquad}lll}
\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} \\
\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 A'' & \Leftrightarrow & \mathcal{A} \mid (\mathcal{A}' \mid A'') & (\mathcal{A} \mid\mid \mathcal{A}') \mid\mid A'' & \Leftrightarrow & \mathcal{A} \mid\mid (\mathcal{A}' \mid\mid 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} \vee \mathcal{A}') \mid \mathcal{A}'' & \Leftrightarrow & (\mathcal{A} \mid \mathcal{A}'') \vee (\mathcal{A}' \mid \mathcal{A}'') & (\mathcal{A} \wedge \mathcal{A}') \mid\mid \mathcal{A}'' & \Leftrightarrow & (\mathcal{A} \mid\mid \mathcal{A}'') \wedge (\mathcal{A}' \mid\mid \mathcal{A}'') \\
\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}'] \\
\exists x.\eta[\mathcal{A}] & \Leftrightarrow & \eta[\exists x.\mathcal{A}] \ (x \neq \eta) & \forall x.\eta[\Rightarrow \mathcal{A}] & \Leftrightarrow & \eta[\Rightarrow \forall x.\mathcal{A}] \ (x \neq \eta) \\
\exists x.\eta[\Rightarrow \mathcal{A}] & \Leftrightarrow & \eta[\Rightarrow \exists x.\mathcal{A}] \ (x \neq \eta) & \forall x.\eta[\mathcal{A}] & \Leftrightarrow & \eta[\forall x.\mathcal{A}] \ (x \neq \eta)
\end{array}
$$

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’.

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

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:

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

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 you find $\mathcal{X}$, which satisfies $\mathcal{A}$, hence it is interpreted using logical conjunction. Formally, path interpretation is defined as shown below; path interpretation translates all non-path operators as themselves, as exemplified for $\mathbf{T}$ and $\mid$.

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

4 The Tree Query Language

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

4.1 The query language

A query language should feature the following functionality:

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

The Tree Query Language (TQL) uses the tree logic for binding and selection, and tree building operations to construct the result. Moreover, we allow some tree functions f, chosen from a set Φ of functions of type $tree \to tree$, to appear in the query. For example, we may have a function sum such that: $sum(\mathbf{0}) = 0$; $sum(n[F]) = n$, if n is an integer, and 0 otherwise; $sum(F \mid F') = sum(F) + sum(F')$. In practice, these functions would include user-defined functions written in an external programming language.

TQL queries:

$Q ::=$	query
$from \ \mathcal{X} \vDash \mathcal{A} \ select \ Q$	substitution-collecting query
$\mathcal{X}$	matching variable
$\mathbf{0}$	empty result
$Q \mid Q$	union of results
$\eta[Q]$	nesting of result
$f(Q)$	tree function, for any f in Φ

The $\mathcal{A}$ in *from* $\mathcal{X} \vDash \mathcal{A}$ *select* Q is a formula of the logic that we defined earlier. As in Section 1.1, we abbreviate a query *from* $\mathcal{X} \vDash \mathcal{A}$ *select from* $\mathcal{X}' \vDash \mathcal{A}'$ *select* Q as *from* $\mathcal{X} \vDash \mathcal{A}, \mathcal{X}' \vDash \mathcal{A}'$ *select* Q

A variable V (where V may be either a tree variable $\mathcal{X}$, a label variable x or a recursion variable ξ) is free in a formula $\mathcal{A}$ if one of its occurrences is not in the scope of any $\forall V$, $\exists V$, μV or νV binder in $\mathcal{A}$. We write $FV(\mathcal{A})$ for the set of free variables of $\mathcal{A}$. Free variables in a query are defined as follows. The basic intuition is that in *from* $\mathcal{X} \vDash \mathcal{A}$ *select* Q, the binder $\mathcal{X} \vDash \mathcal{A}$ binds, in Q, those variables that are free in $\mathcal{A}$.

$$
\begin{array}{lll}
FV(\mathcal{X}) & =_{def} & \{\mathcal{X}\} \\
FV(\mathbf{0}) & =_{def} & \{\} \\
FV(Q \mid Q') & =_{def} & FV(Q) \cup FV(Q') \\
FV(m[Q]) & =_{def} & FV(Q) \\
FV(x[Q]) & =_{def} & \{x\} \cup FV(Q) \\
FV(\textit{from } \mathcal{X} \vDash \mathcal{A} \textit{ select } Q) & =_{def} & \{\mathcal{X}\} \cup (FV(Q) \setminus FV(\mathcal{A}))
\end{array}
$$

The semantics of a query is defined with respect to an 'input substitution' σ that binds all the variables that occur free in the query. In an unnested query, the input substitution is used to bind some top-level names, which play the same role as the table names in SQL, or as the persistence roots in object databases. In a nested query, as we see below, the input substitution of the internal query also conveys the values of the variables that have been bound in the outer query.

The semantics of a *from* $\mathcal{X} \vDash \mathcal{A}$ *select* Q query, with respect to an input substitution σ, is defined as follows. We first define the semantics of $\mathcal{X} \vDash \mathcal{A}$ as the set of all substitutions that are defined on the variables that appear free in $\mathcal{A}$ and that extend the input substitution σ. Then, the result of the query is the multiset union of all the information trees that are returned by evaluating the head Q with respect to each of the σ' in the semantics of $\mathcal{X} \vDash \mathcal{A}$.

For example, consider the following query:

$$
\begin{array}{l}
\textit{from ARTICLES} \vDash \textit{author}[\mathcal{X}] \mid \mathbf{T} \\
\textit{select author}[\mathcal{X}]
\end{array}
$$

If *ARTICLES* is bound to the collection in Section 1.1, then the semantics of the binder is the following set of substitutions (where ... stands for the binding of *ARTICLES* to the whole collection):

$$
\{ \quad (\ldots; \mathcal{X} \mapsto \textit{Cardelli}), \quad (\ldots; \mathcal{X} \mapsto \textit{Ghelli}), \quad (\ldots; \mathcal{X} \mapsto \textit{Gordon}), \quad (\ldots; \mathcal{X} \mapsto \textit{Pierce}) \quad \}
$$

Hence the answer to the query is built by taking the union of the four information trees we get for $\textit{author}[\mathcal{X}]$:

$$
\textit{author}[\textit{Cardelli}] \mid \textit{author}[\textit{Ghelli}] \mid \textit{author}[\textit{Gordon}] \mid \textit{author}[\textit{Pierce}]
$$

The semantics of the $\mathcal{X} \vDash \mathcal{A}$ binder is defined to be a set, which means that any two substitutions in $[\![\mathcal{X} \vDash \mathcal{A}]\!]_\sigma$ differ in the value bound to at least one variable.

The semantics of a query is defined in the following table, where σ' *extends* σ means that $dom(\sigma) \subseteq dom(\sigma')$ and that σ and σ' coincide over $dom(\sigma)$. $[\![Q]\!]_\sigma$ is an information tree, and the interpretation is defined under the assumption that $dom(\sigma) \supseteq FV(Q)$.

Hereafter, with a little abuse of notation, we use $\mathbf{0}$, $m[I]$ and $I \mid I'$ to denote the corresponding operators on information trees.

Query semantics

$$
\begin{array}{lll}
[\![\mathcal{X} \models \mathcal{A}]\!]_\sigma & =_{def} & \{\sigma' \,[\!]\, dom(\sigma') = FV(\mathcal{A}) \cup dom(\sigma),\ \sigma' \ extends\ \sigma,\ \sigma(\mathcal{X}) \in [\![\mathcal{A}]\!]_{\sigma'}\} \\
[\![\mathcal{X}]\!]_\sigma & =_{def} & \sigma(\mathcal{X}) \\
[\![\mathbf{0}]\!]_\sigma & =_{def} & \mathbf{0} \\
[\![Q \mid Q']\!]_\sigma & =_{def} & [\![Q]\!]_\sigma \mid [\![Q']\!]_\sigma \\
[\![\eta[Q]]\!]_\sigma & =_{def} & \sigma(\eta)[[\![Q]\!]_\sigma] \\
[\![from\ \mathcal{X} \models \mathcal{A}\ select\ Q]\!]_\sigma & =_{def} & \bigcup_{\sigma' \in [\![\mathcal{X} \models \mathcal{A}]\!]_\sigma} [\![Q]\!]_{\sigma'} \\
[\![f(Q)]\!]_\sigma & =_{def} & f([\![Q]\!]_\sigma)
\end{array}
$$

Query nesting allows us to restructure data. For example, if we want to rearrange papers according to their year of publication, we first select the years, then, for each year $\mathcal{X}$, we list all the papers of that year.

$$
\begin{array}{ll}
from & ARTICLES \models article[Year[\mathcal{X}] \mid \mathbf{T}] \mid \mathbf{T} \\
select & publications\_by\_year[\ Year[\mathcal{X}] \\
& \qquad\qquad \mid\ (from\ \ ARTICLES \models article[Year[\mathcal{X}] \mid \mathcal{Z}] \mid\ \mathbf{T}, \\
& \qquad\qquad\quad select\ article[\mathcal{Z}]\ ) \\
& \qquad\qquad]
\end{array}
$$

4.2 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:

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

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

$$\{(db \mapsto author[m[]],\ \mathcal{X} \mapsto m[],\ \mathcal{Y} \mapsto F) \,[\!]\, F \in \mathcal{IT}\}.$$

Negation creates a similar problem. Consider the following query.

$$
\begin{array}{l}
from\ db \models \neg author[\mathcal{X}] \\
select\ notauthor[\mathcal{X}]
\end{array}
$$

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

$$\{(db \mapsto author[m[]],\ \mathcal{X} \mapsto F) \,[\!]\, F \in \mathcal{IT}, F \not\equiv m[]\},$$

and the query has the following infinite result:

$$\{notauthor[F] \,[\!]\, F \in \mathcal{IT}, F \not\equiv m[]\}.$$

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 [AHV95], 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 semantics and a query processor for them.

Here we explore both solutions. Our semantics already accounts for unsafe queries, since it does not restrict the set of substitutions generated by a binder to be finite, nor does it restrict the query answer to be finite. In Section 5, we define a constraint-based algorithm to process safe queries and both safe and unsafe binders. In Section 6 we define some syntactic sufficient conditions for safety.

5 Query evaluation

In this section we define an algorithm to evaluate queries.

The kernel of the query evaluation problem is binder evaluation. A binder evaluation algorithm takes a binder element $\mathcal{X} \vDash \mathcal{A}$ and a substitution σ, and returns the set of substitutions $[\![\mathcal{X} \vDash \mathcal{A}]\!]_\sigma$. Although this set of substitutions may be infinite, we develop an algorithm that returns a finite representation of such a set, in terms of a finite disjunction of a set of conjunctive constraints over the substitutions.

5.1 Constraints

In this paper, a *constraint* is a finite representation of a possibly infinite set of substitutions over a common domain.

Before describing our implementation of constraints, we list the constant constraints and the constraint operations. The metavariable $R^{\mathbf{V}}$ ranges over constraints with domain $\mathbf{V}$, and $\mathbf{V}$ ranges over sets of variables.

- $\mathbf{0}^{\mathbf{V}}$ is the empty constraint, with domain $\mathbf{V}$, and it represents the empty set of substitutions. We omit the domain when it is clear from the context, or is irrelevant.
- $\mathbf{1}^{\mathbf{V}}$ is the constraint that represents all substitutions over $\mathbf{V}$; the special case $\mathbf{1}^{\emptyset}$ represents the set consisting only of the trivial substitution with empty domain. We omit the domain when it is clear from the context, or is irrelevant.
- $(R^{\mathbf{V}})_{\mathbf{V}'}$, with $\mathbf{V} \subseteq \mathbf{V}'$, is a constraint extension operation, which represents all substitutions that extend a substitution represented by $R^{\mathbf{V}}$, and whose domain is $\mathbf{V}'$. If $\mathbf{V} = \mathbf{V}'$, then $R^{\mathbf{V}} = (R^{\mathbf{V}})_{\mathbf{V}'}$.
- $(V = k)$, where k is a label or a tree, is a constraint that represents the set consisting of the substitution whose domain is $\{V\}$ and that associates V with k.
- $(V \neq k)$, is a constraint that represents the set of all substitutions whose domain is $\{V\}$ and that associate V with a value in Λ (if V is a label variable) or an information tree (if V is a tree variable) that is different from k.
- $R^{\mathbf{V}} \cup R'^{\mathbf{V}'}$ is a constraint that represents the union of the sets of substitutions represented by $(R^{\mathbf{V}})_{\mathbf{V}''}$ and $(R'^{\mathbf{V}'})_{\mathbf{V}''}$, where $\mathbf{V}'' = \mathbf{V} \cup \mathbf{V}'$.

- $R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'}$ is a constraint that represents the intersection of the sets of substitutions represented by $(R^{\mathbf{V}})_{\mathbf{V}''}$ and $(R'^{\mathbf{V}'})_{\mathbf{V}''}$, where $\mathbf{V}'' = \mathbf{V} \cup \mathbf{V}'$. We write $R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'}$ instead of $\cap$ since the domain-extension effect makes this operation quite similar to the relational natural-join operation. When $\mathbf{V}$ and $\mathbf{V}'$ are disjoint, join coincides with labeled cartesian product, and we may use the notation $R^{\mathbf{V}} \times R'^{\mathbf{V}'}$.

The operators $\cup$ and $\bowtie$ inherit many algebraic laws from set union and intersection. We highlight the following ones:

$$\begin{array}{rclrcl} R^{\mathbf{V}} \bowtie \mathbf{0}^{\mathbf{V}'} & = & \mathbf{0}^{\mathbf{V}\cup\mathbf{V}'} & R^{\mathbf{V}} \bowtie \mathbf{1}^{\mathbf{V}'} & = & (R^{\mathbf{V}})_{\mathbf{V}\cup\mathbf{V}'} \\ R^{\mathbf{V}} \cup \mathbf{0}^{\mathbf{V}'} & = & (R^{\mathbf{V}})_{\mathbf{V}\cup\mathbf{V}'} & R^{\mathbf{V}} \cup \mathbf{1}^{\mathbf{V}'} & = & \mathbf{1}^{\mathbf{V}\cup\mathbf{V}'} \end{array}$$

We now discuss our specific implementation of constraints. We build our constraints starting from *simple constraints.* A *simple constraint* r is a tuple of sets S_i, each labeled with a variable identifier V_i, written as $[V_1 \in S_1, \ldots, V_n \in S_n]$, and such that:

1. each V_i is either a label variable or a tree variable, and S_i is, respectively, a set of labels or a set of information trees (represented as info-terms);

2. the set S_i is either a singleton or the complement of a finite set P (a *co-finite* set), denoted by $\overline{P}$ (we may relax this by generalizing singletons to finite non empty sets);

3. all the V_i variables are different.

For each V_i, $r.V_i$ denotes S_i. The set $dom(r) =_{def} \{V_i\}^{i \in I}$ is the domain of the simple constraint.

A *constraint* $R^{\mathbf{V}}$ is a set of simple constraints, each with the same domain $\mathbf{V}$, which is the common domain of all simple constraints in R ($\mathbf{V}$ is needed when R is the empty set). If $r = [V_1 \in S_1, \ldots, V_n \in S_n]$, we say that a substitution σ over $dom(r)$ *respects* r, written $\sigma \models r$, if, for each $i \in 1 \ldots n$, $\sigma(V_i) \in S_i$. A substitution σ *respects* the constraint $R^{\mathbf{V}}$, written $\sigma \models R^{\mathbf{V}}$, if $\exists r \in R$ such that σ respects r. A constraint represents the set of all substitutions that respect it. For example, the constraint $\{[x \in \{a\}, y \in \{a\}]; [x \in \{b\}, y \in \{b\}]\}$ represents the set of substitutions: $\{(x \mapsto a; y \mapsto a); (x \mapsto b; y \mapsto b)\}$ whilst the constraint $\{[x \in \{a\}, y \in \overline{\emptyset}]\}$ represents all the substitutions that map x to a and y to any label.

A simple constraint is *finitary* if no set S_i is co-finite. A constraint is *finitary* if it only contains finitary simple constraints. A constraint represents a finite set of substitutions if and only if it is finitary. Observe that a finitary constraint is a set of tuples with named components, i.e. a relation in the database sense.

We first define the join $r \bowtie r'$ of two simple constraints r, r', whose domains are $\mathbf{V}$, $\mathbf{V}'$. Let $\mathbf{V}'' = \mathbf{V} \cup \mathbf{V}'$. Let r'' be the tuple over $\mathbf{V}''$ such that $r''.V = r.V$ if $V \in \mathbf{V} \setminus \mathbf{V}'$, $r''.V = r'.V$ if $V \in \mathbf{V}' \setminus \mathbf{V}$, and $r''.V = r.V \cap r'.V$ if $V \in \mathbf{V} \cap \mathbf{V}'$. In the last case, $r.V$ and $r'.V$ are either a singleton or a cofinite set; if one is cofinite, or both are, their intersection can be computed as follows (where P and P' range over finite sets)

$$P \cap \overline{P'} \;=\; P \setminus P' \qquad \overline{P} \cap \overline{P'} \;=\; \overline{P \cup P'}.$$

The result $r''.V$ is either a cofinite set (when both $r.V$ and $r'.V$, are cofinite) or a finite set (when either $r.V$, or $r'.V$, is finite), which may be a singleton, or the empty set. If, for some V, $r''.V$ is the empty set, then $r \bowtie r' = \emptyset$ else $r \bowtie r' = \{r''\}$.

We now define our implementation of the basic constraint operations.

- $\mathbf{0}^{\mathbf{V}}$ is $\emptyset^{\mathbf{V}}$, i.e. the empty set of simple constraints with domain $\mathbf{V}$.

- $\mathbf{1}^{\mathbf{V}}$ is the singleton $\{r\}^{\mathbf{V}}$, where r is the simple constraint that associates $\overline{\emptyset}$ with any variable in $\mathbf{V}$.
- $(R^{\mathbf{V}})_{\mathbf{V}'}$ extends the domain of $R^{\mathbf{V}}$ and adds to every simple constraint r in R a field $V \in \overline{\emptyset}$ for every V in $\mathbf{V}' \setminus \mathbf{V}$.
- $(V = k)$ is $\{[V \in \{k\}]\}^{\{V\}}$, which is the singleton containing the simple constraint that represents the substitution that maps V to k, with the domain $\{V\}$.
- $(V \neq k)$ is $\{[V \in \overline{\{k\}}]\}^{\{V\}}$, which is the singleton containing the simple constraint that represents all the substitutions that do not map V to k, with the domain $\{V\}$.
- $R^{\mathbf{V}} \cup R'^{\mathbf{V}'}$ is the set $(R^{\mathbf{V}})_{\mathbf{V}''} \cup (R'^{\mathbf{V}'})_{\mathbf{V}''}$, with domain $\mathbf{V}''$, where $\mathbf{V}'' = \mathbf{V} \cup \mathbf{V}'$: we first extend the domain of both constraints, and then put all the resulting simple constraints together.
- $R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'}$ is the union of $r \bowtie r'$, for all $r \in R$ and $r' \in R'$, with domain $\mathbf{V} \cup \mathbf{V}'$. If $R^{\mathbf{V}}$ and $R'^{\mathbf{V}'}$ are finitary and are represented as two relations, then $R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'}$ is their natural join.

5.2 Evaluation of binders

We now define an algorithm that, given a binder $F \vDash \mathcal{A}$, returns a constraint representing all the substitutions such that $F \vDash_\sigma \mathcal{A}$, and $dom(\sigma) = FV(\mathcal{A})$.

For this purpose we only need to be able to evaluate the primitive logical operators, since the derived ones can be substituted by their definition. However, negation is a very expensive operation, hence this substitution would produce an inefficient evaluation. In practice, a logical optimizer would rather exploit the equivalences of Section 3.2 to push negation to the leaves of the logical expression, obtaining a formula where negation only appears in front of variables, by applying the transformations listed in the following table.

Pushing negation in front of variables

$\neg(\eta[\mathcal{A}])$	$\rightarrow$	$\eta[\Rightarrow \neg\mathcal{A}]$	$\neg(\eta[\Rightarrow \mathcal{A}])$	$\rightarrow$	$\eta[\neg\mathcal{A}]$
$\neg(\mathcal{A} \mid \mathcal{B})$	$\rightarrow$	$\neg\mathcal{A} \mid\mid \neg\mathcal{B}$	$\neg(\mathcal{A} \mid\mid \mathcal{B})$	$\rightarrow$	$\neg\mathcal{A} \mid \neg\mathcal{B}$
$\neg\mathbf{T}$	$\rightarrow$	$\mathbf{F}$	$\neg\mathbf{F}$	$\rightarrow$	$\mathbf{T}$
$\neg\neg\mathcal{A}$	$\rightarrow$	$\mathcal{A}$	$\neg(\eta \sim \eta')$	$\rightarrow$	$\eta \not\sim \eta'$
$\neg(\mathcal{A} \vee \mathcal{B})$	$\rightarrow$	$\neg\mathcal{A} \wedge \neg\mathcal{B}$	$\neg(\mathcal{A} \wedge \mathcal{B})$	$\rightarrow$	$\neg\mathcal{A} \vee \neg\mathcal{B}$
$\neg(\forall x.\mathcal{A})$	$\rightarrow$	$\exists x.\neg\mathcal{A}$	$\neg(\exists x.\mathcal{A})$	$\rightarrow$	$\forall x.\neg\mathcal{A}$
$\neg(\forall \mathcal{X}.\mathcal{A})$	$\rightarrow$	$\exists \mathcal{X}.\neg\mathcal{A}$	$\neg(\exists \mathcal{X}.\mathcal{A})$	$\rightarrow$	$\forall \mathcal{X}.\neg\mathcal{A}$
$\neg(\mu\xi.\mathcal{A})$	$\rightarrow$	$\nu\xi.\neg\mathcal{A}\{\xi \leftarrow \neg\xi\}$	$\neg(\nu\xi.\mathcal{A})$	$\rightarrow$	$\mu\xi.\neg\mathcal{A}\{\xi \leftarrow \neg\xi\}$

For this reason, we now give an algorithm to evaluate a binder which may contain both primitive and derived operators, so that a logical optimizer, invoked before the evaluation algorithm, is free to perform the negation pushing tranformation, as we do in our current implementation, or any transformation that involves the use of the derived operators.

The algorithm takes a tree F and a formula $\mathcal{A}$, and returns an algebraic expression, $(\!|F \vDash \mathcal{A}|\!)$, built in terms of the constraint algebra extended with an if-then-else operator. The resulting algebraic expression can be executed as is, or, in any realistic implementation, may first undergo algebraic and physical optimization.

The algorithm has the following limitations:

- it can only deal with *guarded* recursion, a notion which we define in Section 5.2.5;
- we restrict the label comparison operators, $\sim$, to equality and inequality;
- the interpretation of $\neg\mathcal{A}$ is restricted to formulas where $\sim$ is not used. Hence, in the presence of equality or inequality, we first have to push negation as specified above. Since negation can always be pushed to the leaves, this is not a real limitation.

The algorithm works by structural recursion on $\mathcal{A}$. We divide its presentation into five subsections, presenting, respectively, the simple operators, the iterators, the quantifiers, the equality operator, and the recursive operators; for reasons of space, we do not report here on the implementation of negation, which is described in [CG].

5.2.1 The simple operators

We first define the clauses that deal with the simple operators, i.e. all the operators apart from the iterators $|$ and $||$, the quantifiers, the recursive operators, and the $\sim$ comparisons. The translation maps a pair $F, \mathcal{A}$ to an algebraic expression $(\!| F \vDash \mathcal{A} |\!)$, such that $dom((\!| F \vDash \mathcal{A} |\!)) = FV(\mathcal{A})$.

We do not include negation in the set of "simple operators", but only the special cases $\neg\mathcal{X}$ and $\neg\mathbf{0}$. This is not a limitation, since negation can always be pushed to the leaves, as described in the previous section.

Binder evaluation, the simple operators

$$
\begin{array}{lll}
(\!| F \vDash \mathbf{T} |\!) & =_{def} & \mathbf{1}^{\emptyset} \\
(\!| F \vDash \mathbf{F} |\!) & =_{def} & \mathbf{0}^{\emptyset} \\
(\!| F \vDash \mathcal{A} \vee \mathcal{B} |\!) & =_{def} & (\!| F \vDash \mathcal{A} |\!) \cup (\!| F \vDash \mathcal{B} |\!) \\
(\!| F \vDash \mathcal{A} \wedge \mathcal{B} |\!) & =_{def} & (\!| F \vDash \mathcal{A} |\!) \bowtie (\!| F \vDash \mathcal{B} |\!) \\
(\!| F \vDash \mathcal{X} |\!) & =_{def} & (\mathcal{X} = F) \\
(\!| F \vDash \neg\mathcal{X} |\!) & =_{def} & (\mathcal{X} \neq F) \\
(\!| F \vDash \mathbf{0} |\!) & =_{def} & \mathit{if}\ F \equiv \mathbf{0}\ \mathit{then}\ \mathbf{1}^{\emptyset}\ \mathit{else}\ \mathbf{0}^{\emptyset} \\
(\!| F \vDash \neg\mathbf{0} |\!) & =_{def} & \mathit{if}\ F \equiv \mathbf{0}\ \mathit{then}\ \mathbf{0}^{\emptyset}\ \mathit{else}\ \mathbf{1}^{\emptyset} \\
(\!| F \vDash n[\mathcal{A}] |\!) & =_{def} & \mathit{if}\ F \equiv n[F']\ \mathit{then}\ (\!| F' \vDash \mathcal{A} |\!)\ \mathit{else}\ \mathbf{0}^{FV(\mathcal{A})} \\
(\!| F \vDash n[\Rightarrow \mathcal{A}] |\!) & =_{def} & \mathit{if}\ F \equiv n[F']\ \mathit{then}\ (\!| F' \vDash \mathcal{A} |\!)\ \mathit{else}\ \mathbf{1}^{FV(\mathcal{A})} \\
(\!| F \vDash x[\mathcal{A}] |\!) & =_{def} & \mathit{if}\ F \equiv m[F']\ \mathit{then}\ (x = m) \bowtie (\!| F' \vDash \mathcal{A}\{x \leftarrow m\} |\!)\ \mathit{else}\ \mathbf{0}^{FV(x[\mathcal{A}])} \\
(\!| F \vDash x[\Rightarrow \mathcal{A}] |\!) & =_{def} & \mathit{if}\ F \equiv m[F']\ \mathit{then}\ ((x = m) \bowtie (\!| F' \vDash \mathcal{A}\{x \leftarrow m\} |\!)) \cup (x \neq m) \\
 & & \qquad\qquad\quad \mathit{else}\ \mathbf{1}^{FV(x[\Rightarrow \mathcal{A}])}
\end{array}
$$

$\mathbf{T}$ and $\mathbf{F}$ return the constraint $\mathbf{1}^{\emptyset}$ and $\mathbf{0}^{\emptyset}$, which represent, respectively, success and failure.

The operator $\vee$ is interpreted by the union of the constraints representing two sets of substitutions. If the two constraints have different domains, they have to be extended to the union of the domain, and this makes the resulting substitution infinitary.

The operator $\wedge$ is interpreted by constraint join.

A variable clause just returns the constraint that represents the relevant substitutions.

A $\mathbf{0}$ clause returns a constraint $\mathbf{1}^{\emptyset}$ if F is $\mathbf{0}$, and a constraint $\mathbf{0}^{\emptyset}$ otherwise.

A clause $F \vDash x[\mathcal{A}]$ returns a constraint $\mathbf{0}^{FV(x[\mathcal{A}])}$ if F does not match $m[F']$, for any m. If F matches $m[F']$, the clause is satisfied by those substitutions that associate x with m and satisfy $F' \models \mathcal{A}$. Those substitutions are represented by $(x = m) \bowtie (\!|F' \models \mathcal{A}|\!)$, but they are also represented by the output of the following computation:

$$(x = m) \bowtie (\!|F' \models \mathcal{A}\{x \leftarrow m\}|\!)$$

which yields an equivalent result, but can be computed more efficiently.

A clause $F \models x[\Rightarrow \mathcal{A}]$ returns a constraint $\mathbf{1}^{FV(x[\Rightarrow \mathcal{A}])}$ if F does not match $m[F']$, for any m, since in this case any substitution over $FV(x[\Rightarrow \mathcal{A}])$ would satisfy it. If F matches $m[F']$, the clause is satisfied by those substitutions that associate x with any label that is different from m, represented by $(x \neq m)$, and by those that associate x with m and satisfy $F' \models \mathcal{A}$, which we computed as in the previous case.

5.2.2 Iterators

A clause $\mathcal{A} \mid \mathcal{B}$ is executed by picking every possible decomposition $F' \mid F''$ of F, and by collecting *all* substitutions that satisfy both $F' \models \mathcal{A}$ and $F'' \models \mathcal{B}$. The nesting $\bigcup(\_ \bowtie \_)$ of the algorithm corresponds to the nesting "**there exists** a decomposition $F' \mid F''$ such that **both** $F' \models \mathcal{A}$ **and** $F'' \models \mathcal{B}$ hold".

A clause $\mathcal{A} \mid\mid \mathcal{B}$ is executed by picking every possible decomposition $F' \mid F''$ of F, and by collecting only those substitutions that satisfy either $F' \models \mathcal{A}$ or $F'' \models \mathcal{B}$ for every possible decomposition. The nesting $\bowtie(\_ \cup \_)$ of the algorithm corresponds to the nesting "**for all** decompositions $F' \mid F''$ **either** $F' \models \mathcal{A}$ **or** $F'' \models \mathcal{B}$ hold".

Binder evaluation procedure, the iterators

$$(\!|F \models \mathcal{A} \mid \mathcal{B}|\!) =_{def} \bigcup_{\{F',F'' \,\|\, (F'|F'') \equiv F, F' \in \mathcal{IT}, F'' \in \mathcal{IT}\}} ((\!|F' \models \mathcal{A}|\!) \bowtie (\!|F'' \models \mathcal{B}|\!))$$
$$(\!|F \models \mathcal{A} \mid\mid \mathcal{B}|\!) =_{def} \bowtie_{\{F',F'' \,\|\, (F'|F'') \equiv F, F' \in \mathcal{IT}, F'' \in \mathcal{IT}\}} ((\!|F' \models \mathcal{A}|\!) \cup (\!|F'' \models \mathcal{B}|\!))$$

Iterator evaluation is very expensive. Let $Sons(F)$ be the set of all $l_i[F_i]$ such that $F \equiv l_1[F_1] \mid \ldots \mid l_n[F_n] \mid \mathbf{0}$. Both iterators require the sub-evaluation $(\!|F' \models \mathcal{A}|\!)$ to be repeated once for every subset of $Sons(F)$. However, in most of the queries we are interested in, iterators are used to express conditions like:

$$m[\mathcal{A}] \mid \mathcal{B}$$

In this case, an $F' \mid F''$ decomposition of F may contribute to the result only if $F' \equiv m[G]$, for some G. Otherwise, $(\!|F' \models m[\mathcal{A}]|\!)$ is $\mathbf{0}$, hence $(\!|F' \models m[\mathcal{A}]|\!) \bowtie (\!|F'' \models \mathcal{B}|\!) = \mathbf{0}$, which does not contribute to the external union.

In the same way, if the formula is:

$$m[\Rightarrow \mathcal{A}] \mid\mid \mathcal{B}$$

then for every $F' \mid F''$ decomposition of F where $F' \not\equiv m[G]$, $(\!|F' \models m[\Rightarrow \mathcal{A}]|\!) = \mathbf{1}$, hence $(\!|F' \models m[\Rightarrow \mathcal{A}]|\!) \cup (\!|F'' \models \mathcal{B}|\!) = \mathbf{1}$, thus it gives no contribution to the result of the external join. For this reason, in these cases we can use the following, more efficient, procedure, which becomes even more efficient in the most common cases $m[\mathcal{A}] \mid \mathbf{T}$ and $m[\Rightarrow \mathcal{A}] \mid\mid \mathbf{F}$. (Cases $x[\mathcal{A}] \mid \mathcal{B}$ and $x[\Rightarrow \mathcal{A}] \mid\mid \mathcal{B}$ also admit a similar simplified evaluation procedure.)

Binder evaluation procedure, the iterators, linear cases

$$(\!|F \models m[\mathcal{A}] \mid \mathcal{B}|\!) =_{def} \bigcup_{\{m[F'],F'' \,\|\, (m[F']|F'') \equiv F, F' \in \mathcal{IT}, F'' \in \mathcal{IT}\}} ((\!|F' \models \mathcal{A}|\!) \bowtie (\!|F'' \models \mathcal{B}|\!))$$

$$
\begin{array}{lll}
(\!|F \models m[\mathcal{A}] \mid \mathbf{T}|\!) & =_{def} & \bigcup_{\{m[F'],F'' \,\|\, (m[F']|F'')\equiv F, F'\in\mathcal{IT}, F''\in\mathcal{IT}\}} (\!|F' \models \mathcal{A}|\!) \\
(\!|F \models m[\Rightarrow \mathcal{A}] \mid\mid \mathcal{B}|\!) & =_{def} & \bowtie_{\{m[F'],F'' \,\|\, (m[F']|F'')\equiv F, F'\in\mathcal{IT}, F''\in\mathcal{IT}\}} ((\!|F' \models \mathcal{A}|\!) \cup (\!|F'' \models \mathcal{B}|\!)) \\
(\!|F \models m[\Rightarrow \mathcal{A}] \mid\mid \mathbf{F}|\!) & =_{def} & \bowtie_{\{m[F'],F'' \,\|\, (m[F']|F'')\equiv F, F'\in\mathcal{IT}, F''\in\mathcal{IT}\}} (\!|F' \models \mathcal{A}|\!)
\end{array}
$$

The $m[\mathcal{A}] \mid \mathbf{T}$ and $m[\mathcal{A}] \mid\mid \mathbf{F}$ operations both involve a number of recursive evaluations that is linear in the number of $m[F']$ trees that appear in $Sons(F)$. $m[\mathcal{A}] \mid\mid \mathbf{F}$ is in principle more expensive than $m[\mathcal{A}] \mid \mathbf{T}$, since the results of the sub-evaluations have to be joined together, which is in general more expensive than the union that is used in the $\mid$ case. In practice, long chains of joins can be efficiently computed in many special cases. For example, if no variable is free in $\mathcal{A}$, then $(\!|F' \models \mathcal{A}|\!)$ is either $\mathbf{0}$ or $\mathbf{1}$, and the join becomes a boolean conjunction. If only one variable is free in $\mathcal{A}$, then join reduces to set union (if both operands return a co-finite constraint), test for equality (if both operands return a singleton constraint), or test for $k \in K$ (if one operand returns a co-finite constraint, and the other one returns a singleton constraint).

5.2.3 Quantifiers

Let us define a projection operator on constraints $\Pi_{\mathbf{V}'} R^{\mathbf{V}}$ as follows, where r *extends* r' means that $dom(r') \subseteq dom(r)$, and r and r' coincide over $dom(r')$:

$$\Pi_{\mathbf{V}'} R^{\mathbf{V}} =_{def} \{r' \,\|\, dom(r') = \mathbf{V} \cap \mathbf{V}',\ \exists r \in R,\ r \textit{ extends } r'\}^{\mathbf{V}\cap\mathbf{V}'}$$

By definition, if $F \models_{\sigma;V\mapsto k} \mathcal{A}$ then $F \models_\sigma \exists V.\mathcal{A}$. Hence, $(\!|F \models \exists V.\ \mathcal{A}|\!)$ can be simply computed as $\Pi_{FV(A)\setminus\{V\}}(\!|F \models \mathcal{A}|\!)$.

Remark 1 *The soundness of this implementation relies on the fact that every simple constraint in R actually represents at least one substitution. Otherwise, if for some F, $\mathcal{A}$ we may compute:*

$$(\!|F \models \mathcal{A}|\!) = \{[V \in \emptyset; \mathcal{X} \in \overline{\emptyset}]\}$$

then, our definition of projection would give us the following answer:

$$(\!|F \models \exists V.\ \mathcal{A}|\!) = \{[\mathcal{X} \in \overline{\emptyset}]\}$$

which is wrong, since the constraint $[V \in \emptyset; \mathcal{X} \in \overline{\emptyset}]$ does not represent any substitution, while $\{[\mathcal{X} \in \overline{\emptyset}]\}$ represents all substitutions over $\mathcal{X}$.

The evaluation of universal quantification $\forall V.\ \mathcal{A}$ is a more complex problem, since we should check the formula $\mathcal{A}$ against an infinite set of substitutions for V. However, constraints help us. Suppose we are able to collect in a set K all constants k_i that are 'meaningful' for V, i.e. such that the result of the query would be different depending on whether V is associated with k_i or with k_j. Then, we can just compute the result once for every constraint $\{[x \in \{k_i\}]\}$, and finally once more for "everything else", represented by the constraint $\{[x \in \overline{K}]\}$. In our definition below, E contains one singleton for each constant k_i that appears in all the sets S in the x column of the relation $(\!|F \models \mathcal{A}|\!)$, plus the set $\overline{\{k_1, \ldots, k_n\}}$.

Definition 1

$$
\begin{array}{lll}
const_V(R') & =_{def} & \{m \,\|\, \exists S.\ [V \in S] \in R',\ S \textit{ is finite},\ m \in S\} \\
 & & \cup\ \{m \,\|\, \exists S.\ [V \in \overline{S}] \in R',\ S \textit{ is finite},\ m \in S\}
\end{array}
$$

Binder evaluation procedure, the quantifiers

$$
\begin{array}{lll}
(\!|F \models \exists V.\ \mathcal{A}|\!) & =_{def} & \Pi_{FV(\mathcal{A})\setminus V}(\!|F \models \mathcal{A}|\!) \\
(\!|F \models \forall V.\ \mathcal{A}|\!) & =_{def} & \text{let } R' = (\!|F \models \mathcal{A}|\!),\ K = const_V(R'),\ E = \{\{k\} \mid k \in K\} \cup \{\overline{K}\} \\
 & & \text{in } \ \bowtie_{e\in E}\Pi_{FV(\mathcal{A})\setminus V}(R' \bowtie \{(V=e)\})
\end{array}
$$

5.2.4 Equality

In order to add equality we have to change the set of simple constraints, as follows. The fields of a simple constraint r now associate a variable V to a triple $\langle \mathbf{V}, \mathbf{V}', S\rangle$ where $\mathbf{V}, \mathbf{V}'$ are two disjoint finite subsets of $dom(r)$, and we say that $Eq(V) = \mathbf{V}$, $Diff(V) = \mathbf{V}'$, and $Range(V) = S$. Such a field is satisfied by a substitution σ if $\sigma(V) = \sigma(V')$ for each V' in $Eq(V)$, $\sigma(V) \neq \sigma(V')$ for each V' in $Diff(V)$, $\sigma(V) \in Range(V)$.

A simple constraint r is well formed iff the following properties hold:

$$
\begin{array}{ll}
\forall V \in dom(r). & V \in Eq(V) \\
\forall V \in dom(r). & Eq(V) \cup Diff(V) \subseteq dom(r) \\
\forall V \in dom(r). & Eq(V) \cap Diff(V) = \emptyset \\
\forall V \in dom(r). & V' \in Eq(V) \Rightarrow Eq(V) = Eq(V'),\ Diff(V) = Diff(V'),\ Range(V) = Range(V')
\end{array}
$$

Equality $x = y$ returns the constraint: $[x \mapsto \langle\{x,y\}, \emptyset, \overline{\emptyset}\rangle, y \mapsto \langle\{x,y\}, \emptyset, \overline{\emptyset}\rangle]$. Inequality $x \neq y$ returns the constraint: $[x \mapsto \langle\{x\}, \{y\}, \overline{\emptyset}\rangle, y \mapsto \langle\{y\}, \{x\}, \overline{\emptyset}\rangle]$.

Natural join $r \bowtie r'$ for these extended constraints is computed in two steps. First, we compute the equivalence relation $\equiv$ among variables that is generated by all the $V \in Eq(V)$ constraints of all fields in r and r'. Then, for each equivalence class $\mathbf{V}$ of $\equiv$, and for each V in $\mathbf{V}$, we define r'' as follows:

$$
\begin{array}{lll}
Eq(V) & = & \mathbf{V} \\
Diff(V) & = & \bigcup_{V' \in \mathbf{V}} Diff(V') \\
Range(V) & = & \bigcap_{V' \in \mathbf{V}} Range(V')
\end{array}
$$

This intersection is *unsatisfiable* if, for some V, either $Range(V)$ is empty, or $V \cap V'$ is not empty. If the intersection is unsatisfiable, then $r \bowtie r'$ is the empty set; otherwise, it is the singleton $\{r''\}$.

5.2.5 Evaluation of recursion

In order to add recursion, we add a recursion operator to the language which is used to translate binders. We enrich the relational language with the expression $\mu M.\lambda G.\mathcal{R}$, which can be evaluated by applying the following reduction rule:

$$(\mu M.\lambda G.\mathcal{R})(F) \ \rightarrow \ \mathcal{R}\{G \leftarrow F\}\{M \leftarrow (\mu M.\lambda G.\mathcal{R})\}$$

The recursive expression is well-formed if G is used inside $\mathcal{R}$ as an information tree, and M as a function that maps information trees to constraints (an *information tree evaluation function*). Of course, this recursion operator allows one to write expressions whose evaluation diverges.

We can now define a translation for recursive *guarded* formulas, i.e. formulas where for each subformula $\mu\xi.\mathcal{A}$ and $\nu\xi.\mathcal{A}$, ξ appears in $\mathcal{A}$ only inside a $\eta[\ldots]$ or $\eta[\Rightarrow \ldots]$ context. Guarded formulas are expressive enough to express regular path expressions (but our previous translation has to be modified a little), and, over finite trees, least and greatest fixpoints coincide for guarded formulas.

Informally, to evaluate a guarded recursive formula $\mu\xi.\mathcal{A}$ (or $\nu\xi.\mathcal{A}$) we evaluate $(\!| F \vDash \mathcal{A}(\xi) |\!)$ and, when this reduces to $(\!| F' \vDash \xi |\!)$, we 'recursively' evaluate $(\!| F' \vDash \mathcal{A}(\xi) |\!)$.

More precisely, recursive formulas are translated as follows. We evaluate both μ and ν using the same fixpoint operator since μ and ν coincide over guarded formulas on finite trees.

Binder evaluation, recursive operators

$$
\begin{array}{lll}
(\!| F \vDash M |\!) & =_{def} & M(F) \\
(\!| F \vDash \mu\xi.\mathcal{A} |\!) & =_{def} & (\mu M.\lambda G.(\!| G \vDash \mathcal{A}\{\xi \leftarrow M\} |\!))(F) \\
(\!| F \vDash \nu\xi.\mathcal{A} |\!) & =_{def} & (\mu M.\lambda G.(\!| G \vDash \mathcal{A}\{\xi \leftarrow M\} |\!))(F)
\end{array}
$$

5.3 Evaluation of queries

We can now extend the evaluation procedure to queries. In a *from* $\mathcal{X} \vDash \mathcal{A}$ *select* Q query, the binder is evaluated with respect to a substitution, and yields a constraint, representing a possibly infinite set of extensions to that substitution. If the constraint is finitary, Q is evaluated once for each substitution, and the results are collected; otherwise, a failure is raised. This algorithm evaluates a nested query *from* $\mathcal{X} \vDash \mathcal{A}$, $\mathcal{X}' \vDash \mathcal{A}'$ *select* Q using a nested-loop technique. In practice, more efficient methods can be easily devised.

Query evaluation

$$
\begin{array}{lll}
(\!| \mathcal{X} |\!)_\sigma & =_{def} & \sigma(\mathcal{X}) \\
(\!| \mathbf{0} |\!)_\sigma & =_{def} & \mathbf{0} \\
(\!| Q \mid Q' |\!)_\sigma & =_{def} & (\!| Q |\!)_\sigma \mid (\!| Q' |\!)_\sigma \\
(\!| \eta[Q] |\!)_\sigma & =_{def} & \sigma(\eta)[(\!| Q |\!)_\sigma] \\
(\!| f(Q) |\!)_\sigma & =_{def} & f((\!| Q |\!)_\sigma) \\
(\!| \textit{from } \mathcal{X} \vDash \mathcal{A} \textit{ select } Q |\!)_\sigma & =_{def} & \text{let } R^{\mathbf{V}} = (\!| \sigma(\mathcal{X}) \vDash \sigma(\mathcal{A}) |\!); \\
 & & \text{if } R^{\mathbf{V}} \text{ is \textit{finitary}, then } \bigcup_{\sigma \vDash R^{\mathbf{V}}} (\!| Q |\!)_\sigma \text{ else fail}
\end{array}
$$

A query *from* $\mathcal{X} \vDash \mathcal{A}$ *select* Q is evaluated once for every binding returned by the evaluation of $\mathcal{X} \vDash \mathcal{A}$, which has to return a finite number of bindings.

6 Safety conditions

We define a notion of *restricted queries* such that every restricted query is guaranteed to be safe, i.e. to generate a finite answer.

We first define, by induction on $\mathcal{A}$, two predicates $\mathcal{A}^V$ ($\mathcal{A}$ binds V) and $\mathcal{A}_V$ ($\mathcal{A}$ co-binds V), such that:

$$\text{if } \mathcal{A}^V \text{ then } \Pi_V(\!| F \vDash \mathcal{A} |\!) \text{ is finitary}, \;\; \text{if } \mathcal{A}_V \text{ then } \Pi_V(\!| F \vDash \neg\mathcal{A} |\!) \text{ is finitary.}$$

Then, we use these predicates to characterize a set of restricted queries which are guaranteed to return a finite result. The full process is described in [CG].

7 Extensions for XML

While it is possible to encode XML documents as pure labeled trees, this is not convenient. Hence, the TQL data model has been extended to deal with XML attributes and with the order relation among siblings. The extension is straightforward, and is described in [CG].

8 Comparisons with related proposals

Here we have described 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 [ABS99], we can encode a labeled tree with a relation *Ref(source:OID, label:*Λ*, destination:OID)*. The nodes of the tree are the OIDs 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 a strict order relation on the OIDs such that, in any tuple, the *source* always 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 $a[\mathbf{0}]$" by saying:

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

There are some differences with our approach. First, our logic is 'modal', which means that a formula $\mathcal{A}$ is always about a specific 'subject' that is part of the database currently being queried. First order logic, instead, does not have an implicit subject: one can, and must, name a subject. For example, our modal formula $a[\mathbf{0}]$ 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. Modality can also be the key towards simpler evaluation algorithms. In the modal case, the implicit subject is taken to be a named root of the database, hence information flows from it to the matching variables. In a classical relational logic there is no fixed direction for the flow of information among variables. This is probably one of the reasons why current query languages base their binding mechanism on pattern matching followed by filtering, rather than on a logic. The modal nature of our logic allows us to base binding on the logic itself.

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

- it makes it easy to describe a record both partially ($b[] \mid c[] \mid \mathbf{T}$: contains $b[]$, $c[]$, and possibly more fields) and completely ($b[] \mid c[]$: 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 'everything but the title': $\mathit{paper}[\mathit{title}[\mathbf{T}] \mid \mathcal{X}]$.

The query language we described derives its essential *from-select* structure from set-theoretic comprehension, through the SQL tradition, and this makes it similar to other query languages for semistructured data, such as StruQL [FFLS97, FFK+98], Lorel [AQM+97, GMW99], XML-QL [DMFLS99], and, to some extent, YATL [CDSS98]. An in-depth comparison between the XML-QL, YATL, and Lorel languages is carried out in [FSW+99], and the reported result is that these languages possess striking similarities. This is based on the analysis of thirteen typical queries, and how they are expressed using the three languages plus XQL. We wrote down the same queries in [Ghe]; the result of this comparison is that, for the thirteen queries in [FSW+99], their TQL expression is very similar to the corresponding XML-QL, with a couple of exceptions. First, we do not have a notion of Skolem functions in TQL, although we plan to add a related notion in future graph-oriented versions of the language. XML-QL has Skolem functions, and they are used in a couple of cases in [FSW+99]. The same queries are expressed in [Ghe] with no Skolem functions, and their expression is not difficult. Second, XML-QL does not seem to have a general way of expressing universal quantification, and this problem shows up in the query in Section 3.3 of [FSW+99]. That query 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.

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

The evaluation technique we propose is related to proposals in the field of constraint databases ([Kan95], [Rev98]). The constraint algebra that we define here, however, does not seem to have been studied before.

9 Conclusions and future extensions

We have defined a query language for semistructured data which is based on the ambient logic.

We modeled a semistructured collection as a labeled tree, but many of our techniques may be extended to collections modeled as labeled graphs. We are currently working on this extension, and the preliminary results are encouraging.

In this study we have exploited a small subset of the ambient logic. The ambient logic, and the calculus, also offer operators to specify and perform tree updates [Car99]. Possible connections with semistructured data updates should be explored. In the same way, we plan to explore the connections between the type systems that have been defined for ambient calculi and the typing needs of semistructured data.

An implementation of TQL is currently being carried out at Pisa University. We are exploiting the implementation model described here.

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

[ABS99] 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.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.

[AQM+97] 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.

[BDHS96] 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.

[Car99] L. Cardelli. Semistructured computations. In *Proc. of the Seventh Intl. Workshop on Data Base Programming Languages (DBPL)*, 1999. Available from www.luca.demon.co.uk.

[CDSS98] 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.

[CG] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. available from http://www.di.unipi.it/~ghelli/papers.html.

[CG98] 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.*

[CG00] 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.

[DMFLS99] 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.

[FFK+98] 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.

[FFLS97] 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.

[FSW+99] 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.

[Ghe] G. Ghelli. TQL as an XML query language. available from http://www.di.unipi.it/~ghelli/papers.html.

[GMW99] 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.

[Kan95] P. Kanellakis. Tutorial: Constraint programming and database languages. In ACM, editor, *PODS '95. Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1995, San Jose, California, May 22–25, 1995*, volume 14 of *Proceedings of the ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*, pages 46–53, New York, NY 10036, USA, 1995. ACM Press.

[PMW96] 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.

[Rev98] Peter Z. Revesz. Safe query languages for constraint databases. *ACM Transactions on Database Systems*, 23(1):58–99, March 1998.