A Query Language for Semistructured Data Based on the Ambient Logic *(Extended Abstract)*

Luca Cardelli
Microsoft Research

Giorgio Ghelli
Università di Pisa

Abstract

The ambient logic is a modal logic proposed to describe the structural and computational properties of distributed and mobile computation. The structural part of the ambient logic is, essentially, a logic of labeled trees, hence it turns out to be a good foundation for query languages for semistructured data, much in the same way as first order logic is a fitting foundation for relational query languages.

1 Introduction

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

Semistructured collections, or *semistructured data*, are collections that do not necessarily respect a predefined schema, but where one can still 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 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.

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 foundations, and the wealth of results that have already been produced for the ambient logic, are helpful in the definition of the language semantics and execution model.

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.

$ARTICLES =$

$article[\ author[Cardelli] \mid author[Gordon] \mid title[Anytime\_Anywhere]$
$\quad \mid conference[POPL] \mid year[2000]$
$\quad \mid keyword[Ambient\_Calculus] \mid keyword[Logics]\ ] \mid$

$article[\ author[Cardelli] \mid title[Wide\_Area\_Computation]$
$\quad \mid booktitle[ICALP] \mid year[1999] \mid pages[403\text{-}444] \mid publisher[SV]\ ] \mid$

$article[\ author[Ghelli] \mid author[Pierce] \mid title[Bounded\_Existentials]$
$\quad \mid journal[TCS] \mid year[1998]\ ]$

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

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

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 $paper[author[Cardelli] \mid \ldots] \mid paper[author[Cardelli] \mid \ldots]$, 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*.

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

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 [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).

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:

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

There are some differences with our approach. Firstly, 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.

Apart from modality, another feature of the 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 ($Ref(o1, a, o2)$). Decomposition 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 '$\mathcal{X}$ is everything but the title': $paper[title[\mathbf{T}] \mid \mathcal{X}]$.

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 [FFLS97, FFK+98], Lorel [AQM+97, GMW99], XML-QL [DMFLS99], Quilt [CRF00], and, to some extent, YATL [CDSS98]. An in-depth comparison between the XML-QL, YATL, and Lorel languages is carried out in [FSW+99], based on the analysis of thirteen typical queries. In [Ghe] we wrote down those same queries in TQL; 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, those XML-QL queries that, in [FSW+99], 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 [HH00] 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. Firstly, 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, KLP00]). The operators that we need in order to deal with recursion, nesting ($m[\mathcal{A}]$), and decomposition ($\mathcal{A} \mid \mathcal{A}'$) are, however, new.

2 The Tree Query Language

Our Tree Query Language (TQL) uses a logical formula for binding trees (pieces of the database) to variables, and then uses tree building operations to construct the result.

A query *from* $\mathcal{X} \vDash \mathcal{A}$ *select* Q is always executed with respect to a *context* σ that associates a tree to the variable $\mathcal{X}$.[1] This tree is the database that is the subject of the query. If $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are the free variables of $\mathcal{A}$, the query is executed as follows.

1. Given σ, the binder $\mathcal{X} \vDash \mathcal{A}$ is evaluated and yields the set of all tuples (or *substitutions*) $\sigma' = [\mathcal{X}_1 \mapsto I_1, \ldots, \mathcal{X}_n \mapsto I_1]$ such that $\sigma'(\mathcal{A})$ is satified by $\sigma(\mathcal{X})$ (this notion is formally defined in the Appendix). Referring to Section 1.1, the binder of the query: *from ARTICLES* $\vDash$ $.article[.author[\mathcal{X}_1]]$ *select* $person[\mathcal{X}_1]$ yields the relation (set of tuples): $\{[\mathcal{X}_1 \mapsto Cardelli], [\mathcal{X}_1 \mapsto Gordon], [\mathcal{X}_1 \mapsto Ghelli], [\mathcal{X}_1 \mapsto Pierce]\}$

2. The reconstruction expression Q is evaluated once for every different σ'; the resulting trees are then combined using $|$. In our example, the reconstruction expression $person[\mathcal{X}_1]$ would yield: $person[Cardelli] \mid person[Gordon] \mid person[Ghelli] \mid person[Pierce]$

2.1 Information trees and info-terms

TQL translates *information trees* (informally, edge-labelled trees) into information trees. 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 I is in $\mathcal{IT}$ then the singleton multiset $\{\langle m, I\rangle\}$ is in $\mathcal{IT}$; $\mathcal{IT}$ is closed under binary multiset union. 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 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.

2.2 Formulas and queries

The logic formulas and the query syntax are defined by the following tables. In the first table we also give a semi-formal definition of the matching relation between information trees, contexts and formulas $I \vDash_\sigma \mathcal{A}$ (the fully formal definition, including a proper account of recursion, is given in Appendix A). 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.

Formulas:

$\eta ::=$	label expression	
n	label constant	for any σ, $\sigma(n) = n$
x	label variable	if $\sigma = [\ldots, x \mapsto n, \ldots]$, $\sigma(x) = n$
$\mathcal{A}, \mathcal{B} ::=$	formulas and matching	
$\mathbf{0}$	empty	$I \vDash_\sigma \mathbf{0}$ if I is the empty tree
$\eta[\mathcal{A}]$	nesting	$I \vDash_\sigma \eta[\mathcal{A}]$ if I is $\sigma(\eta)[I']$, and $I' \vDash_\sigma \mathcal{A}$

[1]The context σ includes the entry points into the database, including possibly URLs and local file names.

$\mathcal{A} \mid \mathcal{B}$	decomposition	$I \vDash_\sigma \mathcal{A} \mid \mathcal{B}$ if for some I', I'', I is $I' \mid I''$, and $I' \vDash_\sigma \mathcal{A} \wedge I'' \vDash_\sigma \mathcal{B}$
$\mathbf{T}$	true	$I \vDash_\sigma \mathbf{T}$ is always true
$\neg\mathcal{A}$	negation	$I \vDash_\sigma \neg\mathcal{A}$ if not $I \vDash_\sigma \mathcal{A}$
$\mathcal{A} \wedge \mathcal{B}$	conjunction	$I \vDash_\sigma \mathcal{A} \wedge \mathcal{B}$ if $I \vDash_\sigma \mathcal{A}$ and $I \vDash_\sigma \mathcal{B}$
$\mathcal{X}$	tree variable	$I \vDash_\sigma \mathcal{X}$ if I is $\sigma(\mathcal{X})$
$\exists x.\mathcal{A}$	label quantification	$I \vDash_\sigma \exists x.\mathcal{A}$ if exists $l \in \Lambda$ such that $I \vDash_\sigma \mathcal{A}\{x \leftarrow l\}$
$\exists \mathcal{X}.\mathcal{A}$	tree quantification	$I \vDash_\sigma \exists \mathcal{X}.\mathcal{A}$ if exists $I' \in \mathcal{IT}$ such that $I \vDash_\sigma \mathcal{A}\{\mathcal{X} \leftarrow I'\}$
$\eta \sim \eta'$	comparison	$I \vDash_\sigma \eta \sim \eta'$ if $\sigma(\eta) \sim \sigma(\eta')$; $\sim$ is $=, \neq$, *like*, ...
$\mu\xi.\mathcal{A}$	least fixpoint	$I \vDash_\sigma \mu\xi.\mathcal{A}$ if $I \vDash_\sigma \mathcal{A}\{\xi \leftarrow \mu\xi.\mathcal{A}\}$
ξ	recursion variable	may appear only in positive positions

Derived formulas and abbreviations:

$\mathbf{F}$	$=_{def}$	$\neg\mathbf{T}$	$I \vDash_\sigma \mathbf{F}$: never true
$\mathcal{A} \vee \mathcal{B}$	$=_{def}$	$\neg(\neg\mathcal{A} \wedge \neg\mathcal{B})$	$I \vDash_\sigma \mathcal{A} \vee \mathcal{B}$: $I \vDash_\sigma \mathcal{A}$ or $I \vDash_\sigma \mathcal{B}$
$\forall x.\mathcal{A}$	$=_{def}$	$\neg(\exists x.\neg\mathcal{A})$	$I \vDash_\sigma \forall x.\mathcal{A}$: for all $l \in \Lambda$: $I \vDash_\sigma \mathcal{A}\{x \leftarrow l\}$
$\forall \mathcal{X}.\mathcal{A}$	$=_{def}$	$\neg(\exists \mathcal{X}.\neg\mathcal{A})$	$I \vDash_\sigma \forall \mathcal{X}.\mathcal{A}$: for all $I' \in \mathcal{IT}$: $I \vDash_\sigma \mathcal{A}\{\mathcal{X} \leftarrow I'\}$
$\eta[\Rightarrow \mathcal{A}]$	$=_{def}$	$\neg(\eta[\neg\mathcal{A}])$	$I \vDash_\sigma \eta[\Rightarrow \mathcal{A}]$: if I is $\sigma(\eta)[I']$ then $I' \vDash_\sigma \mathcal{A}$
$\mathcal{A} \mid\mid \mathcal{B}$	$=_{def}$	$\neg(\neg\mathcal{A} \mid \neg\mathcal{B})$	$I \vDash_\sigma \mathcal{A} \mid\mid \mathcal{B}$: if I is $I' \mid I''$ then $I' \vDash_\sigma \mathcal{A}$ or $I'' \vDash_\sigma \mathcal{B}$
$\nu\xi.\mathcal{A}$	$=_{def}$	$\neg(\mu\xi.\neg\mathcal{A}\{\xi \leftarrow \neg\xi\})$	$I \vDash_\sigma \nu\xi.\mathcal{A}$: $I \vDash_\sigma \mathcal{A}\{\xi \leftarrow \nu\xi.\mathcal{A}\}$ (greatest fixpoint)
$.\eta[\mathcal{A}]$	$=_{def}$	$\eta[\mathcal{A}] \mid \mathbf{T}$	$I \vDash_\sigma .\eta[\mathcal{A}]$: for some I'', I is $(\sigma(\eta)[I'] \mid I'')$ and $I' \vDash_\sigma \mathcal{A}$
$!\eta[\mathcal{A}]$	$=_{def}$	$\eta[\Rightarrow \mathcal{A}] \mid\mid \mathbf{F}$	$I \vDash_\sigma !\eta[\mathcal{A}]$: for any I'', if I is $(\sigma(\eta)[I'] \mid I'')$ then $I' \vDash_\sigma \mathcal{A}$

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 a fixed set Φ

We explain some of the logical operators through examples. 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.

We have exemplified the derived 'one-step' operator $.\eta[\mathcal{A}]$ in Section 1.1. While $I \vDash .m[\mathcal{A}]$ means 'one element of I is labeled m, and its content satisfies $\mathcal{A}$', its dual $!m[\mathcal{A}]$ means 'for any element of I which is labeled m, its content satisfies $\mathcal{A}$'. Hence, all papers whose only author (if any) is *Cardelli* can be retrived by the following query (where we use $\mathcal{X} \wedge \ldots$ as an alternative to a nested binder $\mathcal{X} \vDash \ldots$):

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

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

$$\textit{from} \quad \textit{ARTICLES} \vDash \ .\textit{article}[.\textit{author}[.\textit{e-mail}[\mathcal{X}] \vee .\textit{email}[\mathcal{X}]]] \quad \textit{select} \quad \textit{e-mail}[\mathcal{X}]$$

Using recursion, we look for *e-mail* at the current level or, recursively, at any inner nesting level.[2]

$$from \quad ARTICLES \vDash \ \mu\xi.\ .e\text{-}mail[\mathcal{X}] \vee .email[\mathcal{X}] \vee \exists x.\ .x[\xi] \qquad select \quad e\text{-}mail[\mathcal{X}]$$

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.

$$from \quad ARTICLES \vDash \ .article[.y[z] \wedge z \ like \ \%Ghelli\%] \qquad select \quad found[label[y] \mid content[z]]$$

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. Decomposition $Year[\mathcal{X}] \ \mid \mathcal{Z}$ binds $\mathcal{Z}$ to all fields but the year; this decomposition operation is impossible, or difficult, in most of the other query languages.

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

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 [Ghe].

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

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

[2]When every $\mathcal{X}$ is inside a $m[]$ operator, like in this example, recursion is guaranteed to terminate, but still allows complex queries to be written, such as one which evaluates boolean circuits [NS00].

2.3 Safe queries

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

Consider for example the following query:

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

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 substitutions; for example, if $\sigma(db) = \mathit{author}[m[]]$, then $[\![db \vDash \mathcal{X} \vDash \mathcal{A}]\!]_\sigma$ is the infinite set

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

Negation creates a similar problem. Consider the following query.

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

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

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

and the query has the following infinite result:

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

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 3, we define a constraint-based algorithm to process safe queries and both safe and unsafe binders. In [CG] we define some syntactic sufficient conditions for safety.

3 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 $\mathcal{X} \vDash \mathcal{A}$ 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, in the style of [Kan95, KLP00].

More precisely, we specify a translation of binders into expressions of a constraint algebra, and a technique to evaluate the expressions of this algebra. This approach allows the algebraic expressions to be optimized before execution.

3.1 Constraint algebra

In this paper, a *constraint* $R^{\mathbf{V}}$ is a finite representation of a possibly infinite set of substitutions over a common domain $\mathbf{V}$; this set of substitutions is called the *interpretation* of $R^{\mathbf{V}}$ and is denoted by $[\![R^{\mathbf{V}}]\!]$. The metavariables V and $\mathbf{V}$ range, respectively, over variables and over sets of variables. Every variable V may be either an information tree variable $\mathcal{X}$, whose universe $U(\mathcal{X})$ is defined to be $\mathcal{IT}$, or a label variable x, with $U(x) = \Lambda$. We use $U^{\mathbf{V}}$ to denote the set of all substitutions with domain $\mathbf{V}$ that map every variable $V \in \mathbf{V}$ to a value in $U(V)$.

Since a sostitution σ over $V_1, \ldots, V_n$ can be seen as the named tuple $[V_1 := \sigma(V_1), \ldots, V_n := \sigma(V_n)]$, then a set of substitutions over $V_1, \ldots, V_n$ can be seen as a relation, in the database sense, with schema $V_1 : U(V_1), \ldots, V_n : U(V_n)$. Hence, for every constraint $R^{\mathbf{V}}$, its interpretation $[\![R^{\mathbf{V}}]\!]$ can be seen as a relation over $\mathbf{V}$ (generally infinite); we exploit this isomorphism to define a constraint algebra, based on the application of standard relational operators to constraint expressions.

Before describing our implementation of constraints, we define our constraint algebra.

Constraint algebra:

$I ::=$ information tree expression	
$\mathcal{Y}$	information tree variable
I	information tree
$R^{\mathbf{V}} ::=$ constraint expression	
$\mathbf{1}^{\mathbf{V}}$	universe (every substitution over $\mathbf{V}$)
$R^{\mathbf{V}} \cup R'^{\mathbf{V}'}$	binary union
$\bigcup_{\{\mathcal{Y}' \mid \mathcal{Y}'' = I\}} R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''}$	union of $R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''}$ for all decompositions $\mathcal{Y}' \mid \mathcal{Y}''$ of I
$\bigcup_{\{m[\mathcal{Y}'] \mid \mathcal{Y}'' = I\}} R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''}$	union of $R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''}$ for all decompositions $m[\mathcal{Y}'] \mid \mathcal{Y}''$ of I
$(x \mapsto \eta)$	singleton (one column-one row constraint)
$(\mathcal{X} \mapsto I)$	singleton
$\Pi_{\mathbf{V}'} R^{\mathbf{V}}$	projection
$\overline{R^{\mathbf{V}}}$	complement
if $I = \mathbf{0}$ *then* $R^{\mathbf{V}}$ *else* $R'^{\mathbf{V}'}$	test for $\mathbf{0}$
if $I = n[\mathcal{Y}']$ *then* $R^{\mathbf{V}}_{\mathcal{Y}'}$ *else* $R'^{\mathbf{V}'}$	test for $n[\mathcal{Y}']$
if $I = y[\mathcal{Y}']$ *then* $R^{\mathbf{V}}_{\mathcal{Y}',y}$ *else* $R'^{\mathbf{V}'}$	test for $y[\mathcal{Y}']$

We can now state the fundamental equations satisfied by the interpretation of each closed constraint expression. In this table we use the $\cup$, $\times$, Π and $\setminus$ operators of the relational algebra; $[V := k]$ denotes a tuple with a field V with value k (the substitution that only maps V to k).

Semantics of closed constraint expressions:

$$
\begin{array}{lcl}
\llbracket \mathbf{1}^{\mathbf{V}} \rrbracket & = & U^{\mathbf{V}} \\
\llbracket R^{\mathbf{V}} \cup R'^{\mathbf{V}'} \rrbracket & = & (\llbracket R^{\mathbf{V}} \rrbracket \times U^{\mathbf{V}'\setminus\mathbf{V}}) \cup (\llbracket R'^{\mathbf{V}'} \rrbracket \times U^{\mathbf{V}\setminus\mathbf{V}'}) \\
\llbracket \bigcup_{\{\mathcal{Y}' \mid \mathcal{Y}''=I\}} R^{\mathbf{V}} \rrbracket & = & \bigcup_{\{I',I'' \,\rrbracket\, (I'|I'')=I,\ I'\in\mathcal{IT},\ I''\in\mathcal{IT}\}} \llbracket R^{\mathbf{V}}\{\mathcal{Y}' \leftarrow I'\}\{\mathcal{Y}'' \leftarrow I''\} \rrbracket \\
\llbracket \bigcup_{\{m[\mathcal{Y}'] \mid \mathcal{Y}''=I\}} R^{\mathbf{V}} \rrbracket & = & \bigcup_{\{I',I'' \,\rrbracket\, (m[I']|I'')=I,\ I'\in\mathcal{IT},\ I''\in\mathcal{IT}\}} \llbracket R^{\mathbf{V}}\{\mathcal{Y}' \leftarrow I'\}\{\mathcal{Y}'' \leftarrow I''\} \rrbracket \\
\llbracket (\mathcal{X} \mapsto I) \rrbracket & = & \{[\mathcal{X} := I]\} \\
\llbracket (x \mapsto m) \rrbracket & = & \{[x := m]\} \\
\llbracket \Pi_{\mathbf{V}'} R^{\mathbf{V}} \rrbracket & = & \Pi_{\mathbf{V}'} \llbracket R^{\mathbf{V}} \rrbracket \\
\llbracket \overline{R^{\mathbf{V}}} \rrbracket & = & U^{\mathbf{V}} \setminus \llbracket R^{\mathbf{V}} \rrbracket \\
\llbracket \textit{if } I = \mathbf{0} \textit{ then } R^{\mathbf{V}} \textit{ else } R'^{\mathbf{V}'} \rrbracket & = & \textit{if } I = \mathbf{0} \textit{ then } \llbracket R^{\mathbf{V}} \rrbracket \textit{ else } \llbracket R'^{\mathbf{V}'} \rrbracket \\
\llbracket \textit{if } I = n[\mathcal{Y}] \textit{ then } R^{\mathbf{V}} \textit{ else } R'^{\mathbf{V}'} \rrbracket & = & \textit{if } I = n[I'] \textit{ then } \llbracket R^{\mathbf{V}}\{\mathcal{Y} \leftarrow I'\} \rrbracket \textit{ else } \llbracket R'^{\mathbf{V}'} \rrbracket \\
\llbracket \textit{if } I = y[\mathcal{Y}] \textit{ then } R^{\mathbf{V}} \textit{ else } R'^{\mathbf{V}'} \rrbracket & = & \textit{if } I = m[I'] \textit{ then } \llbracket R^{\mathbf{V}}\{y \leftarrow m\}\{\mathcal{Y} \leftarrow I'\} \rrbracket \textit{ else } \llbracket R'^{\mathbf{V}'} \rrbracket
\end{array}
$$

Our algebra deals with a tree I in three different ways. The singleton operator treats I as an atomic value, and stores it into a relation. The operator $\bigcup_{\{m[\mathcal{Y}'] \mid \mathcal{Y}''=I\}} R^{\mathbf{V}}$ considers I as the set of all its binary decompositions $m[I'] \mid I''$, and computes the union of all corresponding values of $R^{\mathbf{V}}\{\mathcal{Y}' \leftarrow I'\}\{\mathcal{Y}'' \leftarrow I''\}$, somehow similarly to the dependent join of the algebras of OQL-like languages [CD92]. The three pattern matching operators "*if* $I =$ .. *then* *else* " treat I as a non-atomic value, allowing its structure to be inspected.

We will also use some derived operators, which are defined as follows.

Constraint algebra, derived operators and abbreviations:

$$
\begin{array}{llll}
\mathbf{0}^{\mathbf{V}} & =_{def}\ \overline{\mathbf{1}^{\mathbf{V}}} & R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'} & =_{def}\ \overline{\overline{R^{\mathbf{V}}} \cup \overline{R'^{\mathbf{V}'}}} \\
\bowtie_{\{\mathcal{Y}' \mid \mathcal{Y}''=I\}} R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''} & =_{def}\ \overline{\bigcup_{\{\mathcal{Y}' \mid \mathcal{Y}''=I\}} \overline{R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''}}} & \bowtie_{\{m[\mathcal{Y}'] \mid \mathcal{Y}''=I\}} R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''} & =_{def}\ \overline{\bigcup_{\{m[\mathcal{Y}'] \mid \mathcal{Y}''=I\}} \overline{R^{\mathbf{V}}_{\mathcal{Y}',\mathcal{Y}''}}} \\
(x \not\mapsto \eta) & =_{def}\ \overline{(x \mapsto \eta)} & (\mathcal{X} \not\mapsto I) & =_{def}\ \overline{(\mathcal{X} \mapsto I)} \\
R^{\mathbf{V}} \times R'^{\mathbf{V}'} & =_{def}\ R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'} (V \cap V' = \emptyset) & (R^{\mathbf{V}})_{\mathbf{V}'} & =_{def}\ R^{\mathbf{V}} \times \mathbf{1}^{\mathbf{V}'\setminus\mathbf{V}}
\end{array}
$$

By this definition, $R^{\mathbf{V}} \bowtie R'^{\mathbf{V}'}$ is interpreted by the standard relational natural join.

3.2 Constraint algebra implementation

We now discuss our implementation of constraint expressions. We build our constraints starting from *simple constraints*. A *simple constraint* r is a tuple of sets S_i, each labeled with a different variable identifier V_i, written as $[V_1{:=}\, S_1, \ldots, V_n{:=}\, S_n]$, and such that: each V_i is either a label variable or a tree variable, and S_i is, respectively, a set of labels or a set of information trees (represented as info-terms); the set S_i is either a singleton or the complement of a finite set P (a *co-finite* set), denoted by $\overline{P}$ 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 $\mathbf{V}$ ($\mathbf{V}$ is needed when $R^{\mathbf{V}}$ is the empty set). A constraint represents a set of substitutions. For example, the constraint $\{[x{:=}\{a\}, y{:=}\overline{\emptyset}]\}$ 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, i.e. a finite relation, if and only if it is finitary.

We 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 simple constraint with domain $\mathbf{V}''$ such that $r''.V = r.V$ for $V \in \mathbf{V} \setminus \mathbf{V}'$, $r''.V = r'.V$ for $V \in \mathbf{V}' \setminus \mathbf{V}$, and $r''.V = r.V \cap r'.V$ for $V \in \mathbf{V} \cap \mathbf{V}'$. 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): $P \cap \overline{P'} = P \setminus P'$ and $\overline{P} \cap \overline{P'} = \overline{P \cup P'}$. If, for some V, $r''.V$ is the empty set, then we set $r \bowtie r' = \emptyset$; otherwise $r \bowtie r' = \{r''\}$.

We now define our implementation of the constraint operations. $\mathbf{0}^{\mathbf{V}}$ is $\{\}$, i.e. an empty set of simple constraints. $\mathbf{1}^{\{V_1,\ldots,V_n\}}$ is the singleton $\{[V_1 := \overline{\emptyset}, \ldots, V_n := \overline{\emptyset}]\}$, which represents all σ with domain $\{V_1, \ldots, V_n\}$. $(R^{\mathbf{V}})_{\mathbf{V}'}$ adds $V_1 := \overline{\emptyset}, \ldots, V_n := \overline{\emptyset}$ to every simple constraint r in $R^{\mathbf{V}}$, if $\{V_1, \ldots, V_n\} = \mathbf{V}' \setminus \mathbf{V}$. $(V \mapsto k)$ is $\{[V := \{k\}]\}$, the singleton containing the simple constraint that represents the substitution that maps V to k, with the domain $\{V\}$. $(V \not\mapsto k)$ is $\{[V := \overline{\{k\}}]\}$, 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 $\mathbf{V} \cup \mathbf{V}'$. $\Pi_{\mathbf{V}'} R^{\mathbf{V}}$ is the set $\Pi_{\mathbf{V}'} R^{\mathbf{V}} = \{r' \;\|\; dom(r') = \mathbf{V} \cap \mathbf{V}',\ \exists r \in R^{\mathbf{V}}.\ r \textit{ extends } r'\}$, where r *extends* r' means that $dom(r') \subseteq dom(r)$, and that r and r' coincide over $dom(r')$.[3] *if* $I = y[\mathcal{Y}]$ *then* $R^{\mathbf{V}}$ *else* $R'^{\mathbf{V}'}$ is evaluated as $R^{\mathbf{V}}\{y \leftarrow m\}\{\mathcal{Y} \leftarrow I'\}$ if $I = m[I']$ for some m, I', and is evaluated as $R'^{\mathbf{V}'}$ otherwise. *if* $I = \mathbf{0}$ *then* $R^{\mathbf{V}}$ *else* $R'^{\mathbf{V}'}$ and *if* $I = n[\mathcal{Y}]$ *then* $R^{\mathbf{V}}$ *else* $R'^{\mathbf{V}'}$ are evaluated in a similar way. $\bigcup_{\{\mathcal{Y}' | \mathcal{Y}'' = I\}} R^{\mathbf{V}}$ is the union of $R^{\mathbf{V}}\{\mathcal{Y}' \leftarrow I'\}\{\mathcal{Y}'' \leftarrow I''\}$ for every $(I'|I'') = I$. $\bowtie_{\{\mathcal{Y}' | \mathcal{Y}'' = I\}} R^{\mathbf{V}}$ is the natural join of $R^{\mathbf{V}}\{\mathcal{Y}' \leftarrow I'\}\{\mathcal{Y}'' \leftarrow I''\}$ for every $(I'|I'') = I$.

3.3 Evaluation of binders

We now define an algorithm that, given a pair I, $\mathcal{A}$ (where no recursion variable is free in $\mathcal{A}$) returns a constraint representing all the substitutions σ such that $I \in [\![\mathcal{A}]\!]_\sigma$ and $dom(\sigma) = FV(\mathcal{A})$.[4] We will show how to exploit this algorithm to compute the semantics of a binder $\mathcal{X} \vDash \mathcal{A}$.

In principle, we may consider formulas $\mathcal{A}$ that only contain the primitive logical operators, since the derived ones can be always substituted by their definition. However, negation is a very expensive operation, hence this approach would produce an inefficient evaluation. In practice, a logical optimizer would rather try and push negation to the leaves of the logical expression, obtaining a formula where negation only appears in front of variables. 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 an information tree I and a formula $\mathcal{A}$, and returns an algebraic expression, $(\!| I \vDash \mathcal{A} |\!)$, built in terms of the constraint algebra. 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. First, it can only deal with *guarded* recursion (see [CG]). Second, we restrict the label comparison operators, $\sim$, to equality and inequality. Third, 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

[3] The soundness of this implementation relies on the fact that every simple constraint r in $R^{\mathbf{V}}$ is satisfiable, i.e. r actually represents at least one substitution. Otherwise, if, e.g., $\{[V := \emptyset; \mathcal{X} := \overline{\emptyset}]\}$ were a legitimate constraint, then we would get: $\Pi_{\mathbf{V}'}\{[V := \emptyset; \mathcal{X} := \overline{\emptyset}]\} = \{[\mathcal{X} := \overline{\emptyset}]\}$ which is wrong, since the constraint $[V := \emptyset; \mathcal{X} := \overline{\emptyset}]$ does not represent any substitution, while $\{[\mathcal{X} := \overline{\emptyset}]\}$ represents all substitutions over $\mathcal{X}$.

[4] $FV(\mathcal{A})$ is the set of the free label variables and tree variables of $\mathcal{A}$; the binders are $\forall$ and $\exists$.

negation can always be pushed to the leaves, this is not a real limitation.

The algorithm works by structural recursion on $\mathcal{A}$; we do not report here on the implementation of negation, recursion, and universal quantification, which are described in [CG].

We inductively define a function $\llparenthesis I \vDash \mathcal{A} \rrparenthesis$ which maps a pair $I, \mathcal{A}$ to an algebraic expression, such that $dom(\llparenthesis I \vDash \mathcal{A} \rrparenthesis) = FV(\mathcal{A})$. This function transforms the mapping from substitutions to information trees that is defined by a logical formula, into the inverse mapping, from information trees to substitutions, which is needed to evaluate a query. Formally, it enjoys the following property: $I \in [\![\mathcal{A}]\!]_\sigma \;\Leftrightarrow\; \sigma \in [\![\llparenthesis I \vDash \mathcal{A} \rrparenthesis]\!]$ for any $\mathcal{A}$ with no free relation variable $\mathcal{Y}$.

Binder evaluation

$$
\begin{array}{ll}
\llparenthesis I \vDash \mathbf{T} \rrparenthesis & =_{def}\ \mathbf{1}^{\emptyset} \\
\llparenthesis I \vDash \mathbf{F} \rrparenthesis & =_{def}\ \mathbf{0}^{\emptyset} \\
\llparenthesis I \vDash \mathcal{A} \vee \mathcal{B} \rrparenthesis & =_{def}\ \llparenthesis I \vDash \mathcal{A} \rrparenthesis \cup \llparenthesis I \vDash \mathcal{B} \rrparenthesis \\
\llparenthesis I \vDash \mathcal{A} \wedge \mathcal{B} \rrparenthesis & =_{def}\ \llparenthesis I \vDash \mathcal{A} \rrparenthesis \bowtie \llparenthesis I \vDash \mathcal{B} \rrparenthesis \\
\llparenthesis I \vDash \mathcal{X} \rrparenthesis & =_{def}\ (\mathcal{X} \mapsto I) \\
\llparenthesis I \vDash \neg\mathcal{X} \rrparenthesis & =_{def}\ (\mathcal{X} \not\mapsto I) \\
\llparenthesis I \vDash \mathbf{0} \rrparenthesis & =_{def}\ \mathit{if}\ I = \mathbf{0}\ \mathit{then}\ \mathbf{1}^{\emptyset}\ \mathit{else}\ \mathbf{0}^{\emptyset} \\
\llparenthesis I \vDash \neg\mathbf{0} \rrparenthesis & =_{def}\ \mathit{if}\ I = \mathbf{0}\ \mathit{then}\ \mathbf{0}^{\emptyset}\ \mathit{else}\ \mathbf{1}^{\emptyset} \\
\llparenthesis I \vDash n[\mathcal{A}] \rrparenthesis & =_{def}\ \mathit{if}\ I = n[\mathcal{Y}]\ \mathit{then}\ \llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis\ \mathit{else}\ \mathbf{0}^{FV(\mathcal{A})} \\
\llparenthesis I \vDash n[\Rightarrow \mathcal{A}] \rrparenthesis & =_{def}\ \mathit{if}\ I = n[\mathcal{Y}]\ \mathit{then}\ \llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis\ \mathit{else}\ \mathbf{1}^{FV(\mathcal{A})} \\
\llparenthesis I \vDash x[\mathcal{A}] \rrparenthesis & =_{def}\ \mathit{if}\ I = y[\mathcal{Y}]\ \mathit{then}\ (x \mapsto y) \bowtie \llparenthesis \mathcal{Y} \vDash \mathcal{A}\{x \leftarrow y\} \rrparenthesis\ \mathit{else}\ \mathbf{0}^{FV(x[\mathcal{A}])} \\
\llparenthesis I \vDash x[\Rightarrow \mathcal{A}] \rrparenthesis & =_{def}\ \mathit{if}\ I = y[\mathcal{Y}]\ \mathit{then}\ ((x \mapsto y) \bowtie \llparenthesis \mathcal{Y} \vDash \mathcal{A}\{x \leftarrow y\} \rrparenthesis) \cup (x \not\mapsto y)\mathbf{1}^{FV(x[\Rightarrow \mathcal{A}])} \\
\llparenthesis I \vDash \mathcal{A} \mid \mathcal{B} \rrparenthesis & =_{def}\ \bigcup_{\{\mathcal{Y} \mid \mathcal{Y}' = I\}} (\llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis \bowtie \llparenthesis \mathcal{Y}' \vDash \mathcal{B} \rrparenthesis) \\
\llparenthesis I \vDash \mathcal{A} \mid\mid \mathcal{B} \rrparenthesis & =_{def}\ \bowtie_{\{\mathcal{Y} \mid \mathcal{Y}' = I\}} (\llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis \cup \llparenthesis \mathcal{Y}' \vDash \mathcal{B} \rrparenthesis) \\
\llparenthesis I \vDash m[\mathcal{A}] \mid \mathcal{B} \rrparenthesis & =_{def}\ \bigcup_{\{m[\mathcal{Y}] \mid \mathcal{Y}' = I\}} (\llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis \bowtie \llparenthesis \mathcal{Y}' \vDash \mathcal{B} \rrparenthesis) \\
\llparenthesis I \vDash .m[\mathcal{A}] \rrparenthesis & =_{def}\ \bigcup_{\{m[\mathcal{Y}] \mid \mathcal{Y}' = I\}} \llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis \\
\llparenthesis I \vDash m[\Rightarrow \mathcal{A}] \mid\mid \mathcal{B} \rrparenthesis & =_{def}\ \bowtie_{\{m[\mathcal{Y}] \mid \mathcal{Y}' = I\}} (\llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis \cup \llparenthesis \mathcal{Y}' \vDash \mathcal{B} \rrparenthesis) \\
\llparenthesis I \vDash !m[\mathcal{A}] \rrparenthesis & =_{def}\ \bowtie_{\{m[\mathcal{Y}] \mid \mathcal{Y}' = I\}} \llparenthesis \mathcal{Y} \vDash \mathcal{A} \rrparenthesis \\
\llparenthesis I \vDash \exists V.\ \mathcal{A} \rrparenthesis & =_{def}\ \Pi_{FV(\mathcal{A}) \setminus V} \llparenthesis I \vDash \mathcal{A} \rrparenthesis
\end{array}
$$

$\mathbf{T}$ and $\mathbf{F}$ return the constraints $\mathbf{1}^{\emptyset}$ and $\mathbf{0}^{\emptyset}$ representing, 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 clause $I \vDash \mathbf{0}$ returns a constraint $\mathbf{1}^{\emptyset}$ if I is $\mathbf{0}$, and a constraint $\mathbf{0}^{\emptyset}$ otherwise.

A clause $I \vDash x[\mathcal{A}]$ returns a constraint $\mathbf{0}^{FV(x[\mathcal{A}])}$ if I does not match $m[I']$, for any m. If I matches $m[I']$, the clause is satisfied by those substitutions that associate x with m and satisfy $I' \vDash \mathcal{A}$. Those substitutions are represented by $(x \mapsto m) \bowtie \llparenthesis I' \vDash \mathcal{A} \rrparenthesis$, but they are also represented by the output of the computation $(x \mapsto m) \bowtie \llparenthesis I' \vDash \mathcal{A}\{x \leftarrow m\} \rrparenthesis$ which yields an equivalent result, but can be computed more efficiently.

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

m, represented by $(x \not\mapsto m)$, and by those that associate x with m and satisfy $I' \vDash \mathcal{A}$, which we compute as in the previous case.

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

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

General iterator evaluation ($\mid$ and $\mid\mid$) is expensive, because I admits 2^n decompositions $\mathcal{Y}|\mathcal{Y}' = I$, where n is the cardinality of I. However, some common special cases can be done more efficiently. For example, $.m[\mathcal{A}]$ $(m[\mathcal{A}] \mid \mathbf{T})$ requires only to iterate over the $m[\mathcal{Y}]|\mathcal{Y}'$ decompositions of I, which are at most n.

For existential quantification, $(\!|I \vDash \exists V.\ \mathcal{A}|\!)$ can be computed as $\Pi_{FV(A)\setminus\{V\}}(\!|I \vDash \mathcal{A}|\!)$, because $F \vDash_\sigma \exists V.\mathcal{A}$ if, for some k, $F \vDash_{\sigma;V\mapsto k} \mathcal{A}$.

3.4 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.

The notation $\sigma(\mathcal{A})$ stands for $\mathcal{A}$ where every tree variable $\mathcal{X}$ in $dom(\sigma)$ is substituted with the formula $F(\sigma(\mathcal{X}))$, and every label variable $x \in dom(\sigma)$ is substituted by $\sigma(x)$. The formula $F(I)$ is defined as: $F(\mathbf{0}) = \mathbf{0}$, $F(I \mid I') = F(I) \mid F(I')$, $F(m[I]) = m[F(I)]$.

Query evaluation

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

4 Conclusions and future extensions

In the full paper [CG] 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 also define a minor extension of the TQL data model and language in order to deal with some XML features, i.e. XML attributes and the order relation among siblings.

Our query language is based on a tree logic, 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, based on the implementation model we described. The current prototype can be used to query XML documents accessible through files or through web servers.

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.

[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.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In M. Stonebraker, editor, *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, volume 21(2) of *SIGMOD Record*, pages 383–392, New York, NY, USA, June 1992. ACM Press.

[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.

[CRF00] 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.

[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.

[HH00] 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.

[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.

[KLP00] G. Kuper, L. Libkin, and J. Paredaens. *Constraint Databases.* 2000.

[NS00] 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.

A Formal semantics of TQL

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 I is in $\mathcal{IT}$ then the singleton multiset $\{\langle m, I\rangle\}$ is in $\mathcal{IT}$; $\mathcal{IT}$ is closed under binary multiset union. An info-term F denotes an information tree, as follows.

Semantics of info-terms

$$
\begin{array}{lll}
[\![\mathbf{0}]\!] & =_{def} & \{\{\}\} \\
[\![m[F]]\!] & =_{def} & \{\langle m, [\![F]\!]\rangle\} \\
[\![F' \mid F'']\!] & =_{def} & [\![F']\!] \cup [\![F'']\!] \qquad \text{where } \cup \text{ is multiset union}
\end{array}
$$

Formulas denote sets of information trees, as follows.

Semantics of formulas

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

$$I \vDash_\sigma \mathcal{A} \quad =_{def} \quad I \in [\![\mathcal{A}]\!]_\sigma$$

Queries denote information trees, as follows (observe that info-terms are just a subset of queries).

Semantics of queries

$$[\![\mathcal{X} \vDash \mathcal{A}]\!]_\sigma \quad =_{def} \quad \{\sigma' \parallel dom(\sigma') = FV(\mathcal{A}) \cup dom(\sigma),\ \sigma' \ extends\ \sigma,\ \sigma(\mathcal{X}) \in [\![\mathcal{A}]\!]_{\sigma'}\}$$

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