Spatial Tree Logics to reason about Semistructured Data

Giovanni Conforti and Giorgio Ghelli

Dipartimento di Informatica
Università di Pisa, Pisa, Italy

Abstract. The *Ambient Logic* is a modal logic proposed to describe the structural and computational properties of distributed and mobile computations. The static part of the Ambient Logic is, essentially, a spatial logic for unordered labeled trees, hence it turns out to be a good foundation for expressing properties of tree-shaped data (i.e. semistructured data and XML). The Tree Query Language (TQL) is a query language for semistructured data based on the Ambient Logic. TQL uses Ambient Logic formulas to express properties of data that will be collected using a sort of pattern-matching mechanism. Reasoning about constraints for semistructured Data and XML is an active branch of research and has several applications including query optimizations and data integration; it is crucial for validity, type inference, and static type checking. Furthermore the interaction of schema and constraints is both theoretically and practically interesting (in particular for XML), and is currently a largely unexplored area. This paper introduces Spatial Tree Logics (STL, fragments of the logic of TQL) as formalisms for semistructured data constraint, type and query specification and reasoning. As an example, a decidable STL fragment is presented, and it is used to solve some specific decision problems for semistructured data.

1 Introduction

In the era of Web services and Web applications there is a tremendous need for database-like functionality to efficiently provide and access data on the Web. But Data on the Web is free-evolving and ever-changing, and it has various shapes and forms. For these reasons much of the traditional framework of database theory needs to be reinvented in the Web scenario. The database community is performing an intensive work in this direction. The self-describing and irregular structure of data on the Web has been formalized by *semistructured data* (SSD for short). Several schema definition languages for SSD and XML have been proposed. The problem of *constraint specification* (generalization of the classical dependencies to the SSD and XML framework) has been addressed. Many query languages for semistructured data and XML have been studied, but their expressivity is not easy to characterize.

Understanding interaction of schema, constraints and queries for SSD and XML is a very important issue that encloses several current topics of research.

For this reason a formal environment that combines constraints, types and query expressions is an interesting perspective. Using this unified formal framework we can reason about:
(i) **Schema vs. constraints:** Types and constraints decision problems (and their interactions) can be studied in a unified view of schema and constraints. In particular it is possible to study the impact of schema specification on standard constraint decision problems. (ii) **Constraints vs. queries:** Query optimization guided by constraints can be studied. (iii) **Schema vs. queries:** A static type-system can be obtained by combining query and type expressions.

The main problem of such an environment is the expressive power of the formalism we want to use. There is the usual trade-off between expressivity of the formalism and decidability of the resulting language. A well-studied candidate formalism is the Ambient Logic [15, 14]. The idea of using Ambient Logic to describe properties of SSD is due to previous works of Cardelli and Ghelli [13, 11] and to the experience we gained during the implementation of the TQL query language [17, 16].

TQL is a query language for SSD that uses spatial tree logic formulas to express properties of the expected result. The logic used in TQL is very expressive and allows us to express complex types, constraints, and queries, giving us, for types and constraints, an expressive power that is higher than the one of other proposals [18, 4]. However, if the full power of the logic is used, every aspect of static query analysis (correctness, containment, subtyping...) becomes undecidable, since validity of a tree-logic formula is undecidable in general. On the other side, many decidable subsets of the logic can be defined, which are expressive enough to encode known type and constraint systems.

The purpose of this paper is twofold: (i) to provide a gentle introduction to Spatial Tree Logics (STL for short) showing that STL are "good" formalisms for SSD types, constraints and query specification (Section 2); (ii) to present a propositional decidable STL fragment that is sufficient, in some specific cases, to reason about SSD types and constraints (Section 3).

Related work Description logics has been proposed as a unifying formalism to formulate queries and to perform automated reasoning for supporting modeling and query answering; see, for example [10]. Modal logics has been proposed for some of these tasks as well, for example in [2]. The ambient logic differs from the other modal logics mainly because of the presence of the separation operator $A \mid B$, described in the next section. This operator is strictly related to the logical operators recently proposed by Peter O'Hearn, John Reynolds, and others, to reason about shared mutable data structures [21, 20]. A precise characterization of the effect of this operator on the expressive power of the logic is still open to research.

2 Spatial Logics for Trees

The term *Spatial Logics* has been recently used to refer to logics equipped with the separation operator $A \mid B$, described below.

Spatial logics are emerging as an interesting tool to describe properties of several structures. Models for spatial logics include computational structures such as heaps [21, 20], trees [11], graphs [12], concurrent objects [8] as well as process calculi such as the π-calculus [6, 7] and the Ambient Calculus [15, 14].

In [13, 11, 9] spatial logics describing trees are presented. These logics are based on the spatial fragment of the Ambient Logic, i.e. on Ambient Logic without time modalities, and can be used to reason about tree-shaped semistructured data. Here we introduce the data model, the connectives, and the main problems (model checking and validity) of the STL used in TQL [13, 15].

Data Model The spatial fragment of the Ambient Logic is, essentially, a spatial logic describing finite unordered labeled trees, which are called *information trees* in [13].

An information tree (over a label set Λ) is an unordered finite tree whose edges[1] are labeled over Λ. The syntax for information trees is the following:

Syntax for information trees (i.t.'s for short)

$F, F' ::=$	
$\mathbf{0}$	the empty tree consisting of a single root node
$n[F]$	a single edge tree labeled $n \in \Lambda$ leading to the subtree represented by F
$F \mid F'$	the tree obtained by root-merging the trees represented by F and F'

Using the notation above, $a[\, b[\mathbf{0}]|c[\mathbf{0}]\,] \mid a[\mathbf{0}]$ represents a tree with two edges, both labeled by a, carrying $b[\mathbf{0}]|c[\mathbf{0}]$ and the empty tree as children. In this model the composition operator $F \mid F'$ is commutative, hence i.t.'s are unordered.

The description of a tree in the syntax given above is not unique. For example the expressions $F \mid F'$ and $F' \mid F$ represent the same tree; similarly, the expressions $\mathbf{0} \mid \mathbf{0}$ and $\mathbf{0}$ represent the same empty tree. We consider two expressions F and F' equivalent when they represent the same tree, writing $F \equiv F'$. This relation is a congruence and can be fully axiomatized as follows:

$F \mid F' \equiv F' \mid F$	commutativity
$(F \mid F') \mid F'' \equiv F \mid (F' \mid F'')$	associativity
$F \mid \mathbf{0} \equiv F$	neutral element

The information tree data model is a simple *unordered* abstraction of tree-shaped SSD and XML data. Appendix A shows an information tree that represents an XML file with bibtex-like contents.

TQL Spatial Tree Logic We now present TQL Spatial Tree Logic. Hereafter, A and B denote formulas of a spatial tree logic, and x, $\mathcal{X}$, and ξ, denote label, i.t., and recursive variables, respectively. We stipulate that the scope of binders $\forall$, $\exists$, and μ, extends as much as possible to the right, so that $\forall x.A \Rightarrow B$ means $\forall x.(A \Rightarrow B)$.

[1] Edge-labeled information trees can be equivalently described as node-labeled forests, as done in [17].

TQL Logical Formulas:

$A, B ::=$	formula
$\mathbf{0}$	the empty information tree
$n[A]$	single-edge i.t.'s labeled n and leading to a subtree satisfying A
$A \mid B$	i.t.'s splittable into two i.t.'s satisfying A and B respectively
$\mathbf{T}$	all information trees
$\neg A$	all i.t.'s not satisfying A
$A \wedge B$	i.t.'s satisfying A and B
$\mathcal{X}$	the i.t. bound to the variable $\mathcal{X}$
$x[A]$	as $n[A]$, when n is the value bound to the variable x
$\exists x.A_x$	quantification over label variables
$\exists \mathcal{X}.A_{\mathcal{X}}$	quantification over i.t. variables
ξ	recursive variable, only occurring positively
$\mu\xi.A_\xi$	least fixpoint, describing recursive properties

As a first example, the formula $\exists x.\ x[\mathbf{T}]$ describes all single-edged i.t.'s (they can lead to everything because $\mathbf{T}$ matches every i.t.'s). Another example is $n[\mathbf{T}] \mid n[\mathbf{T}]$, describing all i.t.'s made up of exactly two edges labeled n. This formula differs from $n[\mathbf{T}] \wedge n[\mathbf{T}]$, that is equivalent to $n[\mathbf{T}]$, describing a single-edge i.t. labeled n. More interestingly the formula $n[A] \mid \mathbf{T}$ describes all i.t.'s that can be splitted into two sub-trees, one satisfying $n[A]$ and the other satisfying $\mathbf{T}$, that is all i.t.'s containing (at least) an n edge whose content is described by A. We can use this concept to write a simple formula asserting that "there is at least one *book* edge, leading to at least one *author*, containing exactly one edge labeled *Ghelli*, leading to nothing":

$$book[author[Ghelli[\mathbf{0}]] \mid \mathbf{T}] \mid \mathbf{T}$$

In general, formulas of this spatial logic can combine connectives talking about the structure with standard propositional connectives. For example the following formula says that "there exists at least a non empty *book* that does not contain any edge labeled *unpublished*".

$$book[\neg\mathbf{0} \wedge \neg(unpublished[\mathbf{T}] \mid \mathbf{T})] \mid \mathbf{T}$$

Quantification is defined as usual, with label variables x ranging over the infinite set of all possible labels Λ, and i.t. variables X ranging over the infinite set of all i.t.'s.

Least fixpoint positive recursion ($\mu\xi.\ A$) can be intuitively understood as the limit of the sequence of formulas obtained by repeatedly substituting ξ with the full formula $\mu\xi.\ A$. For example, the recursive formula $\mu\xi.\ \mathbf{0} \vee n[\xi]$ is equivalent to $\mathbf{0} \vee n[\mathbf{0} \vee n[\ldots]]$, hence describes a unary tree labeled n. In the formal semantics, each formula denotes the set of trees that satisfy the formula, and the denotation $\mu\xi.\ A$ is defined as a minimal fixpoint. For example, $\mu\xi.\ \mathbf{0} \vee (A \mid \xi)$ denotes the least set S of trees such that $\mathbf{0} \in S$ and, if $F \in S$ and F' satisfies A, then $F \mid F' \in S$. Hence, $\mu\xi.\ \mathbf{0} \vee (a[\mathbf{0}] \mid \xi)$ denotes the set that contains the empty tree and all the one-level trees that have a finite set of outgoing edges all labeled a. The formal semantic definition of this logic is presented in [13].

Model checking *Model checking* is the problem of deciding whether the i.t. F satisfies the closed formula A ($F \vDash A$). An example of model checking is:

$$book[year[1999] \mid author[Ghelli] \mid \ldots] \mid book[\ldots] \vDash book[author[Ghelli[\mathbf{0}]] \mid \mathbf{T}] \mid \mathbf{T}$$

which verifies whether the term at the left hand side of $\vDash$ contains a *book* containing an *author* edge leading to the singleton *Ghelli*.

Model checking is decidable for TQL Spatial Tree Logic. (Model checking is not decidable for the full Ambient Logic [15], since it contains an operator that allows one to express the validity of an arbitrary formula as a model-checking problem.)

Query answering *Query answering* is the problem of collecting a finite representation of the possibly infinite set of all the substitutions σ for the free variables in A such that $F \vDash A\sigma$. Query answering reduces to model-checking when A is closed. Query answering is decidable for the TQL logic, and the TQL system implements a query-answering algorithm, that constitutes the kernel of the TQL query engine. (In TQL the set of substitutions returned by a query-answering problem can be infinite since we do not adopt an active-domain semantics.)

Validity *Validity* is the problem of checking whether a formula is *valid*, i.e. it is satisfied by every finite i.t.. A dual question is *satisfiability*: "is there any finite i.t. satisfying the formula?". The two problems are equivalent for all the logics we consider here, since all of them are closed by negation, and have the property that A is valid iff $\neg A$ is not satisfiable: $(\forall F.\ F \vDash A) \Leftrightarrow \neg(\exists F.\ F \vDash \neg A)$. Most reasoning problems about SSD can be rephrased as validity problems in TQL logic. Unfortunately, validity of TQL logic is undecidable.

In [9] a decidable propositional logic for information trees is presented. This logic consists of propositional primitives (conjunction, implication, and falsity), the spatial primitives of our logic (composition $A \mid B$, location $n[A]$, and empty tree $\mathbf{0}$), and two new spatial primitives: *guarantee* $A \triangleright B$ and *placement* $A@n$. These connectives, already investigated in [15], can be used to express specifications about the properties of the model after it is placed in a surrounding context. In this logic validity and satisfaction problem are, in fact, equivalent and in [9] it is shown how to decide them by model checking and, alternatively, by deduction in a sequent calculus. The decidable logic we present in Section 3 is a fragment of [9] with a very mild extension, an operator ($\%[A]$) to express properties of subtrees regardless the label.

Motivating example The scenario includes a semistructured data source D (e.g. an XML file) on the Web, a schema S (e.g. a DTD or an XML Schema), and a set of integrity constraints C (e.g. some key constraint) specified in some constraint language. Suppose to translate S in an equivalent[2] spatial tree logic formula A_S and similarly C in another formula A_C. The unsatisfiability of A_S

[2] Since our current logic is unordered, all the reasonings (e.g. equivalence) we refer to are up to document order.

corresponds to the well-known problem of *schema emptiness*, and the satisfiability of A_C corresponds to the problem of *constraint consistency*. The satisfiability of the legitimate logic formula $A_S \wedge A_C$ corresponds to another interesting and little investigated problem: *constraint consistency in presence of a schema*.

Other useful schema decision problems that could be investigated in an unified logic include *schema equivalence*, *schema inclusion* and *schema disjointness*. Another classical decision problem is the *constraint implication*: given that some constraints are known to hold, does it follow that some other constraint is necessarily satisfied? This corresponds in our logic to the validity of an implication formula $A_C \Rightarrow A'_C$. Similarly the *constraint implication in presence of schema* corresponds to a $A_S \wedge A_C \Rightarrow A'_C$ where A_S specifies the schema, and A_C and A'_C are constraints. Constraint implication is important, among other things, in data integration, database normalization, and in query optimization.

The *validation problem* of the data source D w.r.t. a schema S and a set of constraints C becomes, in our logic environment, a satisfaction problem of the form $D \vDash A_S \wedge A_C$ in the Ambient Logic style. A generalization of this satisfaction problem dealing with free variables is the core of TQL binding mechanism. This allow us to perform checking of data properties (i.e. validation of constraints) by executing TQL yes/no queries encoding the satisfaction relation.

Our scenario includes also the consumer side of the Web: a user that queries the data source D. The user may ignore schema and constraints over the data source. When the user expresses her query with a tree logic formula B, a static analysis on $A_S \wedge B$ can be performed to check whether the query conforms to the schema. If that formula is unsatisfiable, we can statically state the emptiness of the query result avoiding query execution. In a similar way we can use constraints on D that are known to hold, eventually combined with the schema.

Reasoning on constraints, type and queries can be used as a query optimization tool. As a trivial example we have the following query binding expression: "Bind all the books in the bibliography file that contain at least an author element to X". If we know by the schema (or by a known constraint) that "All books have at least an author", the previous query is equivalent to the cheaper one "Bind all the books in the bibliography file to X".

More interesting examples are optimizations based on key constraint implications and combinations of key constraints and inclusion constraints with the schema. More generally, given the formulas A_S, A_C, and B representing respectively the logic formalization of the schema, the set of validated constraints, and the query, we can substitute B with every cheaper query B' such that $A_S \wedge A_C \Rightarrow (B \Leftrightarrow B')$. Of course the query execution cost depends upon the physical layer and the available index structures. Therefore a complete optimization tool should combine such logical rewritings with standard physical optimization techniques.

2.1 Expressing Schema and Types in TQL Logic

Schema and types of SSD are, essentially, structural properties that a given data source must satisfy. STL formulas describe structural properties of infor-

mation trees, hence STL can be used to express the essence of common schema formalisms for SSD (XML DTDs, XDuce Types [18], and XSD Schema [1]).

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

the type	*a generic element of the type*
$\mathbf{0}$	the empty tree
$\mathbf{S} =_{def} \exists x.\ x[\mathbf{0}]$	a data leaf (a String)
$A \mid B$	an A (i.e. an element of A) next to a B
$A \vee B$	either an A or a B
$n[A]$	an edge n leading to an A
$A^* =_{def} \mu\xi.\ \mathbf{0} \vee (A \mid \xi)$	a finite multiset of zero or more A's
$A^+ =_{def} A \mid A^*$	a finite multiset of one or more A's
$A? =_{def} \mathbf{0} \vee A$	an A, or an empty tree
$\mathbf{T}$	anything

This formulas can be used to express types like the following, borrowed from [19]:

```
type  Addrbook = addrbook[Person*]
type  Person   = person[Name, Addr, Tel?]
type  Name     = name[String]
type  Addr     = addr[String]
type  Tel      = tel[String]
```

We express the type statement `Abook: Addrbook` as the following constraint specification:

$$Abook \vDash addrbook[\ person[\ name[\mathbf{S}] \mid addr[\mathbf{S}] \mid tel[\mathbf{S}]?\ ]^*\ ]$$

Recursive types can be translated in our logic using the modal recursion operator $\mu\xi.A$. For example, the following recursive type

```
type Section = section[intro[String], Section*, conc[String]]
```

can be expressed with: $section[\mu\xi.\ intro[\mathbf{S}] \mid section[\xi]^* \mid conc[\mathbf{S}]\ ]$. We remark that the data model is unordered so this constraint does not imply any order among tags.

As usual, mutual recursion can be encoded as well. For example the type:

```
type Part = part[name[String], Subpart*]
type Subpart = subpart[material[String], Part*, Subpart*]
```

with `Part` as root, can be represented by the formula:

$$\mu\xi_{Part}.\ part[name[\mathbf{S}] \mid \mu\xi_{Subpart}.\ subpart[material[\mathbf{S}] \mid \xi^*_{Part} \mid \xi^*_{Subpart}]]$$

We are still investigating the expressive power of this logic fragment, but we claim that we can express standard XML type systems (i.e., all types expressible by unranked tree automata), of course in an unordered setting.

2.2 Expressing Constraints in TQL Logic

While types constrain the shape of data, it is often useful to constrain the values as well; the canonical examples are key constraints and referential integrity constraints. The SSD constraints that have emerged are normally expressed using *path constraints*, logical statements whose atoms are expressions of the form $r(x, y)$ where r is a regular expression over the set Λ of labels. Intuitively, $r(x, y)$ states that y can be reached from x following the path whose labels spell a word in r. Inclusion constraints can be expressed easily as path constraints in the following way: $p \subseteq q =_{def} \forall x.p(root, x) \Rightarrow q(root, x)$. Key constraints are slightly more complex to express as path constraints, since they also require a notion of equality, as discussed below.

Inclusion and Inverse Constraints To exemplify inclusion constraints we borrow from [5] the typical student-course use case. We first define two types $StudentT$ and $CourseT$, and consider a document CS that contains a set of students and of courses:

$$StudentT =_{def} student[SSN[\mathbf{S}] \mid name[\mathbf{S}] \mid taking[\mathbf{S}]^*]$$
$$CourseT =_{def} course[cno[\mathbf{S}] \mid title[\mathbf{S}] \mid taken\_by[\mathbf{S}]^*]$$
$$CS \vDash StudentT^* \mid CourseT^*$$

The fact that every course taken by a student has to be a valid course, is expressed by the path inclusion constraint $student.taking \subseteq course.cno$ over the data source CS.

We can express the same constraint in TQL logic as follows, where we use the path-notation $.n[A]$ as an abbreviation of $n[A] \mid \mathbf{T}$ (observe that we need here universal quantification over information trees, defined as $\forall\mathcal{X}.A =_{def} \neg\exists\mathcal{X}.\ \neg A$).

$$CS \vDash \forall\mathcal{X}.\ .student[.taking[\mathcal{X}]] \Rightarrow .course[.cno[\mathcal{X}]]$$

The constraints the that the two relationship $student.taking$ and $course.taken$ are symmetric is a canonical example of inverse constraint. It can be expressed in TQL logic as:

$$CS \vDash \forall C.\forall S.\ .student[.SSN[S] \wedge .taking[C]] \Leftrightarrow .course[.cno[C] \wedge .taken\_by[S]]$$

Key Constraint As done by Buneman et al. [4], we can express a notion of *relative* keys. Assume you have a set of books whose type, expressed as in the previous section, is:

$$BOOKS \vDash books[\ book[\ chapter[number[\mathbf{S}] \mid content[\mathbf{S}]]*\ ]*\ ]$$

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

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

In this formula we use the | operator with its full power of *separated conjunction*. We are saying that it is impossible to find a book such that, inside that book, we find two *different* subtrees both satisfying $.chapter[.number[\mathcal{X}]]$, for the same value of $\mathcal{X}$, i.e. two *different* chapters with the same number.

This key notion alone (with no schema definition) does not constrain the key to appear in each element. A notion of mandatory key can be expressed in our logic by adding constraints of the form $\neg\ .books[.book[.chapter[\neg\ .number[\mathbf{T}]]]]$: it is impossible to find a book with a chapter that does not have any number.

The fact that TQL model-checking is decidable means that we can check whether a document satisfies such a constraint. The fact that query-answering is decidable means that we can generalize the formula above, by substituting *number* with a variable t, and the resulting query finds the set of all tags such that t is a key for *chapter* relative to *books.book*. The actual TQL code to accomplish this is here (the ugly $ signs are used in TQL to distinguish labels variables from label constants):

$$from\ \ \$BOOKS \vDash forall\ \$X.\neg books.book[\ .chapter[.\$t[\$X]]\ |\ .chapter[.\$t[\$X]]\ ]$$
$$select\ key[t]$$

Foreign-key constraints can be expressed in terms of key constraints combined with inclusion constraints. As an example, consider the following schema, describing a list of books and a list of authors.

$$books[\ book[\ author[auth\text{-}id[\mathbf{S}]]* \mid \mathbf{T}\ ]*\ ]$$
$$\mid authors[\ author[\ id[\mathbf{S}] \mid \mathbf{T}\ ]*\ ]$$

The foreign key constraint specifies that: (i) each author is identified by an *auth-id* (a key constraint); (ii) the referential integrity constraint: *auth-id*'s have to be included into the actual *id*'s of registered authors

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

As a conclusion, TQL logic allows SSD types and constraints to be specified, and, as we exemplified before, TQL allows one to write queries to check *whether* a constraint holds, and even to discover *which* constraints hold. We can use TQL logic to express type and constraint implication or satisfiability problems, but we have no algorithm to decide them. For this reason, we move now to the definition of a decidable sublogic.

3 A Decidable Sublogic

TQL tree logic has a decidable model checking problem, but its validity and satisfiability are not decidable. This means that we can verify whether a document satisfies a type or a constraint, but we cannot, in general, decide those problems that are quantified over documents, such as type inclusion or constraint satisfiability. This problem can be solved by defining sublogics of TQL logic that are expressive enough to specify types and constraints, but whose validity problem is decidable.

We define here such a logic, together with some useful derived operators.

Propositional Logical Formulas:

$A, B ::=$	formula
$\mathbf{0}$	the empty information tree
$\%[A]$	single-edge i.t.'s leading to a subtree satisfying A
$n[A]$	single-edge i.t.'s labeled n and leading to a subtree satisfying A
$A \mid B$	i.t.'s splittable into two i.t.'s satisfying A and B respectively
$\mathbf{T}$	all information trees
$\neg A$	all i.t.'s not satisfying A
$A \wedge B$	i.t.'s satisfying A and B

$A \vee B$	$=_{def}$	$\neg(\neg A \wedge \neg B)$	disjunction
$A \Rightarrow B$	$=_{def}$	$\neg A \vee B$	implication
$A^{\exists}$	$=_{def}$	$A \mid \mathbf{T}$	some component
$A^{\forall}$	$=_{def}$	$\neg((\neg A) \mid \mathbf{T})$	every component
$Tree$	$=_{def}$	$\%[\mathbf{T}]$	exactly 1 branch
$A^{\forall Tree}$	$=_{def}$	$(Tree \Rightarrow A)^{\forall}$	every tree satisfies A

Using these derived connectives we can express a simplified version of non recursive SSD types. It is simplified since Kleene star can only be applied to types that denote a single-edge tree, such as $a[string]$ or $a[A] \vee b[B]$.

the type	*a generic element of the type*
$\mathbf{0}$	the empty tree
$\mathbf{S} =_{def} \%[\mathbf{0}]$	a data leaf
$A \mid B$	an A (i.e. an element of A) next to a B
$A \vee B$	either an A or a B
$n[A]$	an edge n leading to an A
$A^{*} =_{def} A^{\forall Tree}$	a finite multiset of zero or more A's, when $A \Rightarrow Tree$
$A^{\bar{+}} =_{def} A \mid A^{*}$	a finite multiset of one or more A's, when $A \Rightarrow Tree$
$A? =_{def} \mathbf{0} \vee A$	an A or an empty branch
$\mathbf{T}$	anything

Since this logic is decidable, we can reason about type and constraints expressed in this language. In particular, we can prove type equivalence, subtyping, and constraint implication. Consider for example the following type T:

$$addrbook[person[name[\mathbf{S}] \mid addr[\mathbf{S}] \mid tel[\mathbf{S}]?]^{\bar{*}}]$$

where the sub-formula $person[\ldots]^{\bar{*}}$ is equivalent to the regular expression $person[\ldots]^{*}$ because $person[\ldots] \Rightarrow Tree$. The constraint

$$C =_{def} addrbook[\neg(person[name[\mathbf{S}] \mid name[\mathbf{S}] \mid \mathbf{T}]^{\exists})]$$

states that no component of an address book can be a *person* where the tag *name* appears twice. The constraint implication ($T \Rightarrow C$) is equivalent to the validity of the formula ($T \Rightarrow C$), hence belongs to the class of questions we can answer. This logic is too weak, however, to express path and inclusion constraints. We are currently trying to extend it to express decidable logics such as the one studied in [3] without losing decidability.

4 Conclusions and Future Work

TQL logic is a Spatial Tree Logic expressive enough to describe schemas, types, constraints and queries, over unordered tree-shaped SSD and XML.

We have implemented a system that implements model-checking and query answering for this logic, hence can be used to check whether a piece of data satisfies a constraint or belongs to a type, and of course to execute TQL queries (the system is available at *http://tql.di.unipi.it/tql*). This system is also able, in many situations, to rewrite a query to a more efficient one, but the general problem of formula validity is undecidable, hence we cannot use our system to solve problems such as constraint implication or query containment. Hence, we are looking for decidable sublogics of TQL logic that are expressive enough to describe standard types and constraints. This paper performs a first step in this direction, and presents a decidable TQL logic fragment to describe and reason about a type system for tree-structured data that contains that most common type operators. The next obvious step is the expansion of this logic with operators that allow us to express constraints as well, for example by adapting the approach proposed in [2], where a variant of Propositional Dynamic Logic is proposed to reason about path constraints.

References

1. XML schema. Available from http://www.w3c.org, 2000.
2. N. Alechina, S. Demri, and M. de Rijke. A modal perspective on path constraints. *JLC: Journal of Logic and Computation*, (13):1–18, 2003.
3. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. In *Proc. of DBPL 2001*, volume 2397 of *LNCS*. Springer-Verlag.
4. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. of WWW10*, volume 39 of *Computer Networks*, pages 473–487. Elsevier, May 2001.
5. P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Constraints for semistructured data and XML. *SIGMOD Record 30*, 2001.
6. L. Caires and L. Cardelli. A spatial logic for concurrency (Part I). In *Proc. of TACS 2001*, volume 2215 of *LNCS*, pages 1–37. Springer-Verlag, 2001.
7. L. Caires and L. Cardelli. A spatial logic for concurrency (Part II). In *Proc. of CONCUR'02*, volume 2421 of *LNCS*, page 209. Springer-Verlag, 2002.
8. L. Caires and L.Monteiro. Verifiable and executable logic specifications of concurrent objects in L_π. In *Proc. of ESOP'99*, volume 1381 of *LNCS*, pages 42–56. Springer-Verlag, 2001.
9. C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. In *Proc. of ACM SIGPLAN TLDI'03*.
10. D. Calvanese, G. De Giacomo, and M. Lenzerini. Representing and reasoning on XML documents: A description logic approach. *JLC: Journal of Logic and Computation*, 9(3):295–318, 1999.
11. L. Cardelli. Describing semistructured data. *SIGMOD Record, Database Principles Column*, 30(4), 2001.
12. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Proc. of ICALP*, volume 2380 of *LNCS*, page 597. Springer-Verlag, 2002.
13. L. Cardelli and G. Ghelli. A query language based on the ambient logic. In *Proc. of European Symposium on Programming (ESOP), Genova, Italy*, April 2001.

14. L. Cardelli and A. D. Gordon. Ambient logic. Submitted for publication, available from the authors.
15. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. of POPL*. ACM Press, 2000.
16. G. Conforti, O. Ferrara, and G. Ghelli. TQL Algebra and its Implementation (Extended Abstract). In *Proc. of IFIP TCS*. Kluwer Academic Publishers, 2002.
17. G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The Query Language TQL. In *Proc. of WebDB 2002*.
18. H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proc. of WebDB 2000*, volume 1997 of *LNCS*. Springer-Verlag, 2001.
19. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proc. of the ACM ICFP-00*, volume 35.9 of *ACM Sigplan Notices*, pages 11–22, N.Y., September 18–21 2000. ACM Press.
20. Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *In Proc. of CSL*, 2001.
21. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of 17th IEEE Symponsium on Logic in Computer Science*, 2002.

A XML files and information trees

XML files represent nested tree-shaped data and can be easily translated into equivalent (up to document order) information trees. In the following we show a semistructured data source represented in both formalisms. A bibliography XML file:

```
<bib>
  <article>
    <year>2001</year>
    <title>A Web odyssey: from Codd to XML</title>
    <author><name> Victor </name> <surname> Vianu </surname></author>
    <booktitle>Proc. of PODS 2001</booktitle>
    <series>SIGMOD Record</series>
  </article>
  <article>
    <year></year>
    <title>Ambient Logic</title>
    <author>L. Cardelli</author> <author>A. Gordon</author>
    <note>Submitted for publication</note>
  </article>
</bib>
```

and the equivalent information tree:

```
bib[
  article[ year[2001] |
    title[A Web odyssey: from Codd to XML] |
    author[name[Victor] |  surname[Vianu]] |
    booktitle[Proc. of PODS 2001] |
    series[SIGMOD Record]
  ] |
  article[ year[0] |
    title[Ambient Logic] |
    author[L. Cardelli] | author[A. Gordon] |
    note[Submitted for publication]
  ]
]
```