Type Inference for Queries on Semistructured Data*

Extended Abstract

Tova Milo
Tel Aviv University
milo@math.tau.ac.il

Dan Suciu
AT&T Labs
suciu@research.att.com

Abstract

We study the problem of type checking and type inference for queries over semistructured data. Introducing a novel *traces* technique, we show that the problem is difficult in general (NP-complete), but can be solved in PTIME for many practical cases, including, in particular, queries over XML data. Besides being interesting by itself, we show that type inference and the related traces technique have several important applications, facilitating query formulation, optimization, and verification.

1 Introduction

Semistructured data allows data to be given without a schema. Objects may have arbitrary combinations of attributes, different objects may have the same attribute with different types, collections may be heterogeneous, etc. The schematic information is embedded in the data, i.e. objects have the names of their attributes stored with the object. The model has proven successful in a number of applications like data integration [PGMW95], querying biological data [BDHS96], querying the WWW [MMM96], managing Web sites [FFK+98], or as a general purpose data management system [QRS+95]. In all these applications however, data often has some regularity and ignoring the available (possibly partial) schema results in several drawbacks, like efficiency penalties in querying and storing the data, or the loss of semantics for the user, making it hard to formulate queries. Researchers have proposed several new notions of *schemas* to describe the structure for parts of the data: graph schemas [BDFS97], data guides [GW97], unary datalog schemas [NAM97], Schema Definition Language (ScmDL) [BM99], description logics [DGM98]. The various formalisms differ in the kind of restrictions they can impose on the data: these range from a simple upper bound on the objects attributes (in graph schemas [BDFS97]) to arbitrary regular expressions (in ScmDL [BM99] and DTD's of SGML and XML documents [Gol90]) where the amount of structure described can be tuned from fully unknown (equivalent to no schema at all) to a pretty rigid structure.

In this paper we study the interaction between schemas and queries in semistructured data. We consider the most powerful schemas proposed in previous work (ScmDL [BM99]) in conjunction with the most powerful query language features considered for semistructured data, allowing regular path expressions and joins (both reviewed in Section 2). A specific aspect of the interaction that we study here is type checking and type inference. Besides being interesting by itself, we show that it has several important applications, facilitating query formulation, optimization, and verification:

Type checking and inference Functional programming languages distinguish between *type checking* and *type inference* [Mit90, Mit96]. In the former we are given a program (or query) and a type assignment for its variables and are asked whether the type assignment is correct. In the latter we are not given the type assignment but are asked to discover one. Both are well studied problems in functional programming language. In general type checking is easy, most of the hard work lies in type inference. We consider both problems in Section 3. We found that for queries on semistructured data, type checking can be hard too.

*Work partially supported by the USA-Israel Binational Science Foundation and by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities

We first consider *selection queries*. They return sets (or relations) of objects from the input database: they do not construct new data. (Queries constructing new data are considered next). We show that, in general, type inference and checking for selection queries is NP-complete. The high complexity stems from the interaction of regular expressions and joins in the query with untagged union types and unordered data. We consider a number of possible restrictions that are of practical interest, in particular in the context of XML data (http://www.w3.org/TR/REC-xml), and identify some which admit PTIME type checking / inference algorithms (e.g. for join-free queries, or tagged schemas, both over ordered data), and others which are still NP-complete. Our analysis offers a nearly complete picture of which combinations of features lead to PTIME algorithms and which to NP-complete problems (see Table 2). We then show in Section 4 that the above results, and the techniques we developed in this context, have several important applications, as detailed below.

Query formulation Here we take the view that schemas are meant to assist users in writing queries. Schemas are exposed to the users, and the type inference mechanisms are used to signal to the users queries, or query parts, which are inconsistent with the given schema, and furthermore, provide the user with a feedback assisting in query formulation.

Query optimization Alternatively, we take the view that schemas are transparent to the user and used by the optimizer. We assume that the data is stored in a naive manner (as a graph, with no indexes) where the access pattern is traversal from one node to its successors. In a simple setting, the query parts that are inconsistent with the schema may be dropped, resulting in simpler queries. In a more sophisticated setting, we use the type inference techniques to design an *adaptive optimal* evaluation algorithm, which minimizes the number of traversed edges. The algorithm is *adaptive*, in that it adapts its behavior based on the data seen so far, in addition to the schema. It is *optimal* in the sense that no other deterministic algorithm of its class can outperform it.

Data transformation and integration Here we assume that the query specifies a data transformation, and that the schema for the input data is given. The problem is to ensure that the query's output conforms to some other given schema. At a finer level, we may want to derive the schema for the output first, then check that it is subsumed by the required output schema. The importance of this increases with the advent of XML. The new web standard makes it possible for user communities to agree upon a common schema(DTD), then publish and exchange on the Web XML data conforming to that DTD. Several proposals for declarative specification of transformations in semistructured data have been considered in the literature [PAGM96, BDHS96, FFK+98, CDSS97]. We will consider here transformations by means of Skolem functions (an abstraction of the above languages). Given an input schema and a query, we show that, in general, there may not exist an output schema best describing the transformation result, but we identify a particular class of transformations for which such a schema exists.

While our presentation considers most general definitions for semistructured data, schemas, and query languages, we pay special attention to XML, DTD's and query languages for XML. For the latter a few proposals exists: XSL (http://www.w3c.org) and XML-QL [DFF+9u]. We will discuss throughout the paper how our results apply to XML, DTD's, and these query languages.

2 Background

Data Model Our semistructured data model is an ordered version of the *object exchange model* OEM [PGMW95]. Data consists of a collection of *objects*, denoted by oids. Each object has a value which can be either (a) an atomic value, like an int, float, multimedia object, etc., or (b) an *unordered* collection of (label, oid) pairs, or (c) an *ordered* sequence of (label, oid) pairs. labels are drawn from a (possibly infinite) universe $\mathcal{A}$ of label names (strings). In addition there exists a distinguished *root* object, and all objects are reachable from the root. It is customary to visualize such data as a graph, where nodes represent objects and edges are labeled with elements from $\mathcal{A}$. We will use here a textual representation of the data, as defined in Table 1. A data graph is then given by a set of definitions of the form oid=value (atomic value), or oid={E} (unordered collection) or oid=[E] (ordered collection), where E is list of pairs label→Oid. For example o1={a→o2,b→o3} denotes that o1 is an unordered object (with two outgoing edges labeled a,b pointing to o2,o3 resp.), while o2=[a→o4,c→ o5,c→o6] means that o2 is an ordered object. By convention, the first oid defined is the root object, and each oid is defined at most once. We distinguish between referenceable and non-referenceable objects. The former have their name prefixed with &, e.g. &o3, &o45. A non-referenceable object is allowed to occur at most once in the right hand side of a definition.

(The root, if it is non-referenceable, is not allowed to occur in any right hand side.)

Given our special interest in XML data, we illustrate how it can be expressed in our data mode. For example the XML fragment:

```
⟨paper⟩⟨title⟩ A real nice paper ⟨/title⟩
        ⟨author⟩⟨name⟩⟨firstname⟩ John ⟨/firstname⟩
                      ⟨lastname⟩ Smith ⟨/lastname⟩⟨/name⟩
                ⟨email⟩ ... ⟨/email⟩
        ⟨/author⟩
⟨/paper⟩
```

becomes:

```
o1 = [paper → o2];
o2 = [title → o3, author → o4];
o3 = "A real nice paper";
o4 = [name → o5, email → o6];
o5 = [firstname → o7, lastname → o8];
o6 = "...";   o7 = "John";   o8 = "Smith"
```

All objects here are non-referenceable. If we want to share author o4 among several papers, we have to change its name to &o4.

Types We will follow here the formalism of ScmDL [BM99]. Types have a definition which closely resembles that of data. A *schema* consists of a sequence of type definitions each of the form Tid = atomicType, or Tid = {R}, or Tid = [R], where Tid is a type identifier and R is a regular expression over Label→Tid pairs see Table 1. Every schema has a *root* type id, which, by convention, is the first type id listed. Each type id is defined at most once. Type ids are of two kinds: referenceable, and non-referenceable. The former have a name starting with &, like &T5, &T99.

As demonstrated in [BM99], ScmDL allows large flexibility in the structure being described, ranging from very loose (as in Web pages) to rigid structure (as in database sources). Again, we relate types and schemas to XML. The structure of an XML document can be validated by a Document Type Descriptor (DTD). For example, the following could be a DTD for the XML example above:

```
⟨!ELEMENT   Document    (paper*) ⟩
⟨!ELEMENT   title       #PCDATA ⟩
⟨!ELEMENT   paper       (title,(author)*)⟩
⟨!ELEMENT   firstname   #PCDATA ⟩
⟨!ELEMENT   author      (name, email)⟩
⟨!ELEMENT   lastname    #PCDATA ⟩
⟨!ELEMENT   name        (firstname,lastname)⟩
⟨!ELEMENT   email       #PCDATA ⟩
```

It is equivalent to the following schema definition S:

```
DOCUMENT = [(paper→PAPER)*];
PAPER = [title→TITLE.(author→AUTHOR)* ];
AUTHOR = [name→NAME.email→EMAIL];
NAME = [firstnamen→FIRSTNAME.
          lastname→LASTNAME];
TITLE = string;
FIRSTNAME = string;
LASTNAME = string;
EMAIL = string
```

All types in this example are non-referenceable. If we wanted to share author objects, we would have renamed the type name to &AUTHOR.

DTD's can be viewed as schemas where (1) all types are ordered, (2) all types are *tagged*, i.e. there exists a one-to-one correspondence between labels and type ids and an expression label→Tid is allowed only if label and tid are in this correspondence, and (3) all types are non-referenceable (since XML data is tree data). We will denote with DTD- the class of such schemas. In reality DTD's *do* allow references between objects, but do not enforce the type of the target of that reference: a reference is always the union of all types. We denote with DTD+ the more general class of schemas satisfying conditions (1) and (2) above. Hence DTDs are generalizations of DTD- and particular cases of DTD+.

Intuitively, a data graph G *conforms* to a schema S (or, is an *instance* of S) if we can map nodes to type ids s.t. the requirements on the nodes structure in the schema are satisfied. We define this below. Before, we need some notations. Given a regular expression R over some alphabet Σ, we use $lang(R)$ to denote the regular language defined by R. The *unordered* language of R, $ulang(R)$, is a set of finite bags b of elements in Σ s.t. $b \in ulang(R)$ iff there exists some ordering w of the elements in b s.t. $w \in lang(R)$.

Definition 2.1 *Let G be a data graph and S be a schema. We say that G* conforms *to S if there exists a mapping τ from the nodes in G to types in S s.t.:*

1. *τ maps the root node to the root type id.*
2. *if $\&o$ is a referenceable node, then $\tau(\&o)$ is a referenceable type.*
3. *if $o = v$ is a node definition in G (with v an atomic value) then there exists a definition $\tau(o) = atomicType$ in S s.t. v belongs to $atomicType$.*
4. *if $o = \{E\}$ is in G, then there exists a type definition $\tau(o) = \{R\}$ in S such that $\tau(E) \in ulang(R)$; if $o = [E]$ is in G, then there exists a type definition $\tau(o) = [E]$ in S s.t. $\tau(E) \in lang(R)$. The notation $\tau(E)$ means $a_1 \to \tau(o_1)$, $\ldots$, $a_n \to \tau(o_n)$, when E is $a_1 \to o_1$, $\ldots$, $a_n \to o_n$.*

We call such a mapping τ a *type assignment* of G (w.r.t S). Observe that in general several type assignments may be possible for a given data graph: conditions guaranteeing the uniqueness of the assignment are discussed in [BM99]. Checking whether

<table>
<tr><td>DATA GRAPHS</td><td>GraphDef ::= Oid=Node; ...; Oid=Node
Node ::= value | {E} | [E]
E ::= label→Oid, ..., label→Oid</td></tr>
<tr><td>Example</td><td>o1={a→o2, b→o3}; o2=[a→o4,c→o5,c→o6];
o3=3.14; o4="abc"; o5=2.71; o6=6.12</td></tr>
<tr><td>TYPES</td><td>SchemaDef ::= Tid=Type;...;Tid=Type
Type ::= atomicType | {R} | [R]
R ::= (R.R) | (R|R) | (R*) | ϵ | label→Tid</td></tr>
<tr><td>Example</td><td>T1={(a→T2,b→T3)|(d→T4)}; T2=[a→T5,(c→T6)*];
T3=float; T4=int; T5=string; T6=float</td></tr>
<tr><td>PATTERNS</td><td>PatDef ::= nodeVar=Pat; ...; nodeVar=Pat
Pat ::= value | valueVar | {P} | [P]
P ::= L→nodeVar, ..., L→nodeVar
L ::= R | labelVar
R ::= (R.R) | (R|R) | (R*) | ϵ | label | _</td></tr>
<tr><td>Example</td><td>X={a* →Y,(b|(c.d))→U}; Y=[a→Z,(c|d)→V];
U=3.14; V=2.71</td></tr>
</table>

Table 1: Grammars defining data graphs, types, and patterns

a data graph G conforms to a schema S is in general NP-complete, but becomes PTIME for a large class of schemas [BM99], including in particular the tagged schemas: hence conformity for DTD- and DTD+ schemas can be determined in time polynomial in the size of the data graph and the schema.

Finally, we remark that ScmDL allows to use *predicates*, instead of constant labels, in the type definitions. For example one could write AUTHOR = [isName→NAME,...], where isName is a unary predicate on labels. In this extended abstract we chose not to have label predicates in the schemas: all our results (positive and negative) extend to schemas with predicates, e.g. by applying directly the techniques in [AV97]. We defer the treatment of predicates to the full version of the paper.

Patterns and Queries We focus first on *selection queries* that return sets (or relations) of objects from the input data graph: they do not construct new data. (Queries constructing new data will be discussed later). We consider queries of the form:

```
SELECT Var, ..., Var
WHERE PatDef
```

The WHERE clause consists of a series of pattern definitions, basically describing a (sub)graph to be searched in the data. The SELECT clause then projects out on some of the pattern variables. The pattern structure is formally defined in Table 1. Variables are of three kinds: node variables, label variables, and value variables, denoted nodeVar, labelVar, valueVar. Each pattern definition is one of nodeVar = value, nodeVar = valueVar, nodeVar = {P}, or nodeVar = [P], where P is a list of pairs L→nodeVar. L is a regular path expression on labels, or just a label variable. Notice, again, the resemblance to the definition of data graphs and schemas.

We require that each node variable is defined at most once (as in the case of data and type definitions), and that the languages defined by regular path expressions do not contain the empty string (since they are supposed to describe actual, non empty, paths). As before, the first node variable in the pattern definition is called the *root* variable. Among node variables we distinguish between referenceable and non-referenceable. The former have a name prefixed by &, like &X1, &Y4. A non-referenceable variable is allowed to occur at most once in the right hand side of a pattern definition. (The root, if it is non-referenceable, is not allowed to occur in any right hand side.)

For example, the following query Q searches for papers where both Abiteboul and Vianu are authors, with Vianu coming first:

```
SELECT  X1
WHERE   Root = [paper→X1];
        X1 = [author.name.(_*)→X2,
              author.name.(_*)→X3];
        X2 = "Vianu";     X3 = "Abiteboul"
```

Here _ denotes any label, hence (_*) denotes any arbitrary path in the data. To relate our syntax to actual XML query languages, the above query is expressed in, e.g. XML-QL, as

```
WHERE     <paper> $X1 </paper>                    IN Root,
          <author[$i].name.*> Vianu </>           IN $X1,
          <author[$j].name.*> Abiteboul </>       IN $X1,
          $i < $j
CONSTRUCT <result> $X1 </result>
```

Observe that, besides the slightly different syntax, the pattern used in the WHERE clause is very much like ours, except that the values "Vianu" and "Abiteboul" are used directly rather than being associated with distinct variables. (The additional $\$i < \$j$ statement is addressed below). We chose to use patterns with variables everywhere because we found it the most convenient notation for type inference and the related problems studied in this paper. It is straightforward to translate any XML-QL pattern into our pattern notation (and the same for other XML query languages).

Given a query Q with a set of pattern definitions PD and a data graph G, the *result* of applying Q to G consists of a set of *bindings* of pattern variables with nodes, labels, and atomic values from G. Regular expressions in the patterns must then correspond to paths in the data. Before we formally define the semantics of selection queries, we need to define a notion of order among paths in the data graph.

Definition 2.2 *(Order of paths) Given a data graph G and two paths p_1,p_2 in G, we say that p_1* precedes *p_2 if p_1 and p_2 start at the same ordered node v and p_1's first edge is different than that of p_2 and precedes it.*

Given nodes $o_1, \ldots, o_k$ in a graph, and words $w_1, \ldots, w_k \in \mathcal{A}^*$, let P denote the list $w_1 \to o_1, \ldots, w_k \to o_k$. For an unordered node o we say that $\{P\}$ is satisfied at o if there exists paths $P_1, \ldots, P_k$ s.t. P_i goes from o to o_i and it's edges are labeled with w_i. For an ordered node o we say that $[P]$ is satisfied at o if, in addition to the above condition, the paths are ordered. Similarly, if $P' = R_1 \to o_1, \ldots, R_k \to o_k$, with $R_1, \ldots, R_k$ regular expressions, we say that $\{P'\}$ ($[P']$) is satisfied at some node o if there exists words $w_1 \in lang(R_1), \ldots, w_k \in lang(R_k)$ such that $\{R\}$ ($[R]$) is satisfied at o, where $R = w_1 \to o_1, \ldots, w_k \to o_k$.

We are now ready to define the semantics of selection queries.

Definition 2.3 *Let $PD(\bar{x})$ be a set of pattern definition with variables $\bar{x} = (x_1, \ldots, x_n)$, and let G be some data graph. A binding θ from $\bar{x}$ to G is said to satisfy PD if it maps node variables to nodes, label variables to labels, value variables to values, and, in addition:*

1. *θ maps the root node variable to the root node in G.*
2. *if x_i is a referenceable node variable, then $\theta(x_i)$ is a referenceable node.*
3. *if $x_i = v$ is in PD, with v a value, then $\theta(x_i) = v$ is in G.*
4. *if $x_i = x_j$ is in PD, with x_j a value variable, then $\theta(x_i) = \theta(x_j)$ is in G.*
5. *if $x_i = \{P\}$ (or $x_i = [P]$) is in PD, then $\{\theta(P)\}$ (or $[\theta(P)]$) is satisfied at $\theta(x_i)$, where $\theta(P)$ is defined as follows. For $P = L_1 \to x_{j_1}, \ldots, L_k \to x_{j_k}$, $\theta(P)$ is $\theta(L_1) \to \theta(x_{j_1})$, ..., $\theta(L_k) \to \theta(x_{j_k})$, where $\theta(L)$ denotes $\theta(x)$ when L is a label variable x and denotes R when L is a regular expression R.*

A design choice in the semantics of patterns is whether to allow multiple path expressions appearing in one pattern definition to be bound to paths sharing the same first edge. For example, considering the pattern {author→Y, author→Z}, do we allow the two author paths to be bound to the same edge (hence Y and Z bound to the same oid) ? Our choice is that in ordered nodes, the bounded paths must be ordered hence their first edges are disjoint. For the unordered nodes we chose a set-like semantics, allowing paths to overlap in their first edge. (This is also the choice taken by most XML and semistructured data query languages). The other alternative has only minor effects of the complexity results presented in the paper.

Remark: XML query languages like XSL, and XML-QL sometimes allow more flexibility in the specification of order constraints on the paths. For example, in XML-QL one can specify *partial* order among paths, e.g. $[a[i] \to X,\ b[j] \to Y,\ c[k] \to Z,\ d[l] \to U]$, $i < k$ *AND* $j < l$.
In this paper we focus on total orders, i.e. corresponding to $i < j < k < l$. As far as the complexity results below are concerned, the effect of allowing partial orders is the same as the higher of the complexities of ordered or unordered patterns.

In the rest of the paper we assume that the pattern definitions in the query are "connected", i.e. that the root variable transitively refers to all the variables appearing in the pattern.

3 Type Inference for Selection Queries

In the sequel we fix a schema S, and a query Q We define the following problems:
(1) *Type Correctness* (or *Satisfiability*): Does there exist a database G conforming to S on which Q returns a non-empty result?
(2) *Total Type Checking*: Given a type assignment

to all the node and value variables in the query, and a label assignment to the label variables, does there exist a database G conforming to S and a binding of the node/value/label variables with nodes/values/labels from G (respecting the given label assignment), which yields that type assignment?
(3) *(Partial)Type Checking*: Given a type and label assignment of only the variables in the SELECT clause, does there exist a database G conforming to S and a binding of variables with nodes/values/labels from G (respecting the given label assignment) yielding that type assignment?
(4) *Type Inference*: Enumerate all type and label assignments for the variables in the SELECT clause for which (Partial) Type Checking has a positive answer.

For example, consider the schema S and query Q in Section 2. Q is satisfiable for S, but is not satisfiable if evaluated, for instance, w.r.t the schema

DOCUMENT = [(paper→PAPER)*];
TITLE = string;
PAPER = [title→TITLE.author→AUTHOR];
AUTHOR = [name→NAME];
NAME = string

(because it allows a single author). For the original schema, total type checking is positive, e.g., for the type assignment (Root/DOCUMENT, X1/PAPER, X2/LASTNAME,X3/FIRSTNAME) but is negative for the type assignment (Root/DOCUMENT, X1/PAPER, X2/LASTNAME, X3/EMAIL) . (Partial) type checking is positive, e.g., for the type assignment X1/PAPER and is negative for X1/NAME . Finally, type inference here infers a single type, PAPER, for the selected variable X1.

Adapting the taxonomy of [Var82] for query and data complexity, we will study the complexity of these problems in two settings. (1) *Query Complexity:* For a fixed schema S, what is the complexity as a function of the query size? (2) *Combined Complexity:* What is the complexity as a function of both the query size and the schema size ? [1]

Note that, in the type inference problem, the size of the answer may be large, compared to the query and the schema: if the schema is very loose, it may be the case that each of the variables can be associated with most of the types in the schema, so the size of the answer can be up to $O(|Q|^{|S|})$, i.e. exponential in the size of the schema. In this case the complexity is defined in terms of the size of the input *and* the output.

Our general result for these problems is given by the following:

Theorem 3.1 *Both the satisfiability and the type checking (total and partial) problems are in general NP-complete, in query and combined complexity. Type inference can be done in time exponential in the size of the query (and the schema, for combined complexity), and cannot in general be solved in polynomial time (even w.r.t the size of the result in addition to that of the query and schema) unless P=NP.*

Proof: (Sketch) To prove the NP-hardness of the problems we use reduction from the 3SAT problem (testing the satisfiability of 3NF formulas), known to be NP-complete. The completeness is proved using the *traces* technique introduced below in Section 3.4. □

Next we consider restrictions on the query and/or the schema: some combinations result in polynomial time solutions to the above problem. We have two goals here. On the one hand we are interested in finding the largest classes of schemas and queries for which the above problems can be solved in polynomial time (hopefully covering most of the practical common cases). On the other hand we are interested in finding the tightest conditions under which the problems still remain NP-complete. For the schema, we consider the following restrictions:

Ordered schemas: These are schemas where all types are ordered. As a relaxation we consider schemas having ordered types and *homogeneous unordered collections*: the latter are types of the form T = {(a→T')*}. (Other relaxations are briefly considered in the sequel).

Tagged schemas: Consider the relation between labels and type identifiers consisting of pairs (a,T) s.t. the expression a→T occurs at least once in the schema. In a *tagged* schema this relation is one-to-one.

Tree schemas: These are schemas with no referenceable types.

As observed earlier, DTD- schemas are ordered, tagged, tree schemas, while DTD+ are ordered, tagged schemas. On the query side, we will consider the following restrictions.

Projection free: these are queries where *all* variables occur in the SELECT clause.

Constant labels: In a *constant labels* query all the path expressions in the pattern are constant labels and no label variables are used. On a finer

[1] A third kind of complexity is *Schema Complexity*, where only the schema varies. Since in practice this scenario is less common we considered it of less importance and defer a discussion to the full version of the paper.

level we relax the restriction on the path expressions and allow also paths with *Constant suffix:* path expressions of the form $R.l$ where R is some regular expression and l a label constant.

Join free: Given a query Q with a pattern P, we say that a variable x_i in P *refers* to a variable x_j if x_j appears on the right hand side of x_i's definition. A query is *join free* if no variable in it is being referred to multiple times (by one or several variables) or transitively refers to itself. On a finer note we relax this and consider *Bounded joins:* Queries where the number of variables that violate the join-free conditions is bounded by some constant B.

The rational behind these restrictions is the following. Projection-free queries allow reducing the problem of *partial* type checking to the *total* one, which, as shown below, is simpler. The study of constant-labels queries highlights the effect of using regular path expressions relative to the simple traditional constant paths. The study of join-free queries is interesting for several reasons. First, some XML query languages are join-free (XSL). Second, PTIME algorithms for join free queries can be used as an approximation algorithms for languages with joins (like XML-QL)

3.1 Type Correctness (Satisfiability)

We start by considering type correctness. Our results are summarized in Table 2, where we present the effect of the various restrictions on the complexity (both query and combined) of the problem. Each entry in the table is of the form query complexity/combined complexity. NP here means NP-complete. The NP-completeness proofs follow similar lines as in Theorem 3.1. The PTIME results are based on the *Traces* technique described below in Section 3.4.

The upper leftmost result states the NP completeness of the problem in the general case. Naturally, a major factor in the complexity is the use of unordered types: to determine if a pattern variable can be assigned a certain type, all the possible orders of its outgoing paths may need to be considered. It turns out however that order alone does not suffice to reduce the complexity (see leftmost item of line 2). We thus proceed in two directions: (i) We consider possible syntactic restrictions on the query, as illustrated in the rest of lines 2 and 3. In particular observe that the problem becomes polynomial for join-free (and bounded-joins) queries. (ii) We restrict the allowed schemas, requiring types to be tagged. Tagging alone does not suffice, at least for reducing the combined complexity, (see line 4). But together with order it reduces the complexity not only for join-free (bounded joins) queries, but also for all queries with constant-suffix paths. (Hence also for queries using constant labels rather than regular expressions/label variables.) Finally, the rightmost column shows that all the above restrictions are not effective without order.

Relating this to queries over XML data, one can see that the satisfiability of join free XML queries, e.g. queries in XSL, can be tested in PTIME for DTD- and DTD+ schemas. For queries with joins, e.g. XML-QL, constant-suffix is required. Observe that although join over node variables is meaningless for DTD- data (since the instances are trees), joins of label variables are possible and suffice for NP-completeness. If this is disallowed satisfiability is solvable in PTIME (joins on value variables are still allowed).

Remark: We conclude with a comment regarding unordered types. We saw above that allowing certain unordered types, e.g. homogeneous collections, does no harm the complexity. Other unordered types and patterns, e.g. tuple-like, can also be supported. In general, to guaranty low complexity, the number of orders one needs to consider when matching the pattern to the schema should be bounded. For lack of space this is not discussed further here.

3.2 Type checking

We next consider type checking. It turns out that total type checking is simpler than partial. First observe that partial type checking is as difficult as satisfiability since the two problems coincide for boolean queries (i.e. with empty SELECT clause). Furthermore, one can show that all the results in Table 2 hold for partial type checking as well. In contrast we have that

Proposition 3.2 *Total type checking in is PTIME, (both query and combined complexity), for ordered schemas (plus homogeneous collections) and arbitrary queries.*

3.3 Type inference

The results for type inference follow similar classification as that of satisfiability:

- In all cases where satisfiability in Table 2 is NP-complete, type inference can be performed in time exponential in the size of the query (and the schema, for combined complexity), and cannot be done in time polynomial in the query (the schema) and the result, unless P=NP.

Query complexity/ Combined complexity	Arbitrary queries (e.g. XML-QL)	Join free queries (e.g XSL)	Bounded joins queries	Constant labels queries	Constant suffix queries	J.free c.labels queries
Arbitrary schemas	NP/NP	?/NP	?/NP	NP/NP	NP/NP	?/NP
Ordered schemas	NP/NP	P/P	P/P	NP/NP	NP/NP	P/P
Ordered schemas plus homogeneous collections	NP/NP	P/P	P/P	NP/NP	NP/NP	P/P
Tagged schemas	NP/NP	?/NP	?/NP	NP/NP	NP/NP	?/NP
Tagged, ordered schemas (DTD+)	NP/NP	P/P	P/P	P/P	P/P	P/P
Tagged, ordered tree schemas (DTD-)	NP/NP	P/P	P/P	P/P	P/P	P/P
Tagged, ordered schemas plus homogeneous collections	NP/NP	P/P	P/P	P/P	P/P	P/P

Table 2: Summary of Complexity Results for the Type Correctness (Satisfiability) Problem

- Similarly, in all the cases where satisfiability is solvable in PTIME, we prove that type inference can be done in time polynomial in the size of the query (the schema) and the result.

3.4 Traces

In the reminder of this section we briefly describe the technique underlying most of the above results, and in particular the (N)PTIME and EXPTIME algorithms for the problems. As we show in the next section, this technique has also several other important applications.

Simple P-traces We first consider satisfiability. Assume we are given a single, ordered pattern definition P, containing no label variables: $X = [R_1 \to X_1, \ldots, R_k \to X_k]$. Define a *trace* to be a word $Xw_1X_1 \ldots w_kX_k$ in $(\mathcal{A} \cup \{X, X_1, \ldots, X_k\})^*$ (here $w_1, \ldots, w_k \in \mathcal{A}^*$): the set of traces in P, $Tr(P)$, is the language defined by the regular expression $XR_1X_1 \ldots R_kX_k$. Given a data graph G with root o, we say that the trace $Xw_1X_1 \ldots w_kX_k$ *occurs* in G if there exists nodes $o_1, \ldots, o_n$ such that $[w_1 \to o_1, \ldots, w_k \to o_k]$ is satisfied at o: the set of all traces occurring in G is denoted $Tr(G)$. Finally, given a schema S, let $Tr(S) \stackrel{\text{def}}{=} \bigcup\{Tr(G) \mid G \text{ conforms to } S\}$. It is relatively easy to see that P is satisfiable for S iff $Tr(P) \cap Tr(S) \neq \emptyset$. We show in the full version of the paper that $Tr(S)$ is a regular language, for which a regular expression can be constructed in polynomial time from S. Hence, this particular satisfiability problem is in P, since computing the intersection of two regular expressions can be done in polynomial time.

Type checking and inference For type checking or inference we replace the variable symbols with special signs representing the possible types for the variables: For each variable X_i and each type T_j in the schema, we introduce a new symbol $X_i^{T_j}$. The refined definitions of $Tr(P)$ and $Tr(S)$ will now account for all the possible types of nodes at the end of the the k paths. $Tr(P) = \tilde{X}R_1\tilde{X}_1 \ldots R_k\tilde{X}_k$, where each $\tilde{X}_i$ is the regular expression $(X_i^{T_1} \mid X_i^{T_2} \mid \ldots)$. As before, we construct the intersection language $Tr(P) \cap Tr(S)$. Each word in this language represents a possible type assignment s.t. X_i is assigned type T_j iff the symbol $X_i^{T_j}$ appears in the word. To retrieve all possible assignments, it suffices to erase the other symbols (i.e. symbols in $\mathcal{A}$) from the intersection language.

The general case In general we may have several pattern definitions, unordered nodes, joins, and tagged types. We explain below how multiple patterns and unordered types are handled. Joins and tagging are deferred to the full paper. Assume a query with a set of pattern definitions PD, with variables $X, X_1, \ldots, X_k$. A trace is $Xw_1X_1 \ldots w_kX_k$, as before, and the same for $Tr(PD)$ ($Tr(PD) = XR_1X_1 \ldots R_kX_k$, where R_i is the unique regular expression preceding the variable X_i). We only change the definition of $Tr(G)$ to account for the shape of the pattern, and $Tr(S) = \bigcup\{Tr(G)\}$ (as before). Finding a regular expression for $Tr(S)$ is more difficult however (a straightforward construction results in an expression of exponential size). Instead, we construct a definition of $Tr(S)$ bottom up, following the tree structure of the set of pattern definitions. This results in a acyclic, extended[2] context free grammar describing $Tr(S)$, whose size is polynomial in S (and whose expansion is a regular expression of exponential size). Satisfiability reduces to $Tr(P) \cap Tr(S) \neq \emptyset$, as before, and this can be checked in polynomial time. To support unordered nodes, all possible orders and overlapping among the path expressions in the pattern need to

[2]That is, the productions may have regular expressions on the right hand side.

be considered. Rather than defining $Tr(P)$ using a single regular expression, it is now the union of the expressions representing these possible orders, (an exponential number). This exponential blowup can nevertheless be avoided in the case of homogeneous collections: due to the uniformity of the types any arbitrary order can be picked.

4 Applications

We now discuss three applications of type inference in semistructured data. "Applications" should be taken here in a broad sense: applications of interactions between types and queries and of the traces technique developed in this context, rather than direct consequences of the previous complexity results.

4.1 Query formulation and user feedback

Schemas can be used to design user-friendly interfaces for query formulation. Such an undertaking is however beyond the scope of this paper. Here we only show that types can be used to provide user feedback during query formulation. For example consider the Document schema in Section 2 and assume that the user query is

```
Q = SELECT X3
    WHERE Root=[paper.author→X1];
          X1=[_*.name._*→X2 . _*.email→X3];
          X2="Gray";
```

The first and last _* (in _*.name and _*.email) are redundant: they would have made sense if *author* had a more complex nested structure with name and email located deep in the hierarchy. The second _* (in name._*) is not redundant, but _* can only be firstname or lastname. Both informations can be helpful to the user. They can be provided to him as a *feedback query*:

```
Q = SELECT X3
    WHERE Root=[paper.author→X1]
          X1=[name.(firstname|lastname))→X2.
              email→X3];
          X2="Gray";
```

To simplify, we restrict below our discussion to queries with a single ordered join-free pattern definition: the extension to queries with multiple pattern definitions is straightforward (with joins and unorder having the expected effect on the complexity). Given a query Q with a pattern X = [R1→X1,..., Rk→Xk], and a schema S, the *feedback query* Q' has a pattern X = [R1'→X1, ..., Rk'→Xk] s.t. (a) Q and Q' are equivalent on all databases conforming to S, (b) $\mathtt{lang}(\mathtt{Ri}') \subseteq \mathtt{lang}(\mathtt{Ri})$, for all $\mathtt{i} = \mathtt{1} \ldots \mathtt{k}$, and (c) the query is the "minimal" such, i.e. for any other query equivalent to Q on S, with pattern X = [R1"→X1, ..., Rk"→Xk], we have $\mathtt{lang}(\mathtt{Ri}') \subseteq \mathtt{lang}(\mathtt{Ri}'')$ for all $\mathtt{i} = \mathtt{1} \ldots \mathtt{k}$.

It is easy to show, using the *traces* technique described above:

Proposition 4.1 *The feedback query can be computed in PTIME from Q and S.*

Proof: (Sketch) We sketch below the main idea. Consider the P-traces $Tr(P)$ and $Tr(S)$ as in Section 3.4, and let $Tr \stackrel{\text{def}}{=} Tr(P) \cap Tr(S)$. Recall that Tr is a set of words of the form $Xw_1X_1w_2X_2 \ldots w_kX_k$. Then we define Ri' to be the regular language obtained as "the i's projection of Tr", i.e.: $lang(Ri') = \{w_i \mid \exists w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_k,$ s.t. $Xw_1X_1 \ldots w_kX_k \in \mathtt{Tr}\}$ It is easy to check that this is indeed a feedback query. □

4.2 Optimization

Alternatively, schemas and traces can be used by the query processor, to improve the evaluation algorithm. This is a form of semantic optimization. Here we illustrate for a particular computation model. Extensions to other models are beyond the scope of this paper. In our model the data graph is viewed as an ADT with the following edge-traversal operations:

- firstEdge(x), the first (left-most) edge e of the node oid x.
- nextEdge(e), the next edge (right brother) of e, or null when e is last.

Our reference point will be the *naive evaluation strategy*. This searches the graph in a depth-first fashion using the two operations, going downward, rightwards, or backtracking to the previous parent node when the processing of the relevant descendents of a node has been completed. We assume that once backtracked from a node, the algorithm does not return to the node. We focus here on algorithms that, given a schema S, improve over the naive algorithm by pruning some of the search. Our cost function is the number of edges explored by the two traversal operations and the goal is to find an algorithm of the above sort that minimizes the cost function.

Example Before presenting the optimization process we illustrate two techniques for improving over the naive strategy. In the examples below the schema allows only finitely many databases, hence we enumerate the instances rather than give the schema. Also, for brevity, instead of defining separately each instance node, we sometimes merge the definitions

and use a nested notation.
(1) *Downwards pruning:* Consider the query SELECT X WHERE Root=[_.c→X], and a schema S with the following possible instances:

DB1=[a→[c→[]]];
DB2=[a→[d→[]]];
DB3=[b→[d→[]]].

When evaluating the query, if we see a b edge we can stop the search early (and return the empty set).
(2) *Sidewards pruning:* Consider a query SELECT X,Y WHERE Root=[a.b→X, c.d→Y] , and schema with instances:

DB1=[a→[e→[],b→[]], c→h, c→d];
DB2=[a→[e→[],b→[]], c→h, c→h];
DB3=[a→[f→[],b→[]], c→d, c→h];
DB4=[a→[f→[],b→[]], c→h, c→h].

As we search for the first path a.b we will encounter either a e or a f. [3] This "teaches" us how to prune later: in the first case we can prune the search under the first c, in the second case we can prune the access to the second c and backtrack earlier.

Of particular interest are queries over XML data. Since XML data is typically of tree structures, we focus below on join-free queries over DTD- schemas. To simplify, we illustrate the optimization for single pattern queries of the form SELECT X1,...,Xk WHERE Root = [R1→X1, ..., Rk→Xk]. The extension to multiple patterns is straightforward. Other extensions (e.g. to non-tagged, unordered databases, joins, partial variable selection) are beyond the scope of this paper.

We now briefly describe the optimization algorithm. Given a schema S and a query Q, we use traces (similar to those of Section 3.4) to construct a non deterministic automata O accepting a regular language $lang(O) \subseteq (\mathcal{A} \cup \{X\})^*$: here we need a single symbol X standing for any of the variables in the pattern. (The actual construction of the automata is omitted). Using this automata we now modify the naive evaluation algorithm as follows. Let r be the root of the data graph. Rather than searching for outgoing paths corresponding to words in $R_1, \ldots, R_k$, we now look for paths corresponding to words of O: we do the search is a depth first fashion as before, running O's automata against the graph, with one difference – each time that O tells us to look for a X edge, rather than continuing with the children of the current node, we prune the search and move to the next (right) child of the root. Note however that we still insist on a *depth first search*, meaning that all the moves of the above form are in fact deferred to be executed only after the processing of edges below the previous root child is finished.

We will refer to the algorithm above as A_O. The following property guarantees its optimality. Observe that at each point in time the portion of the graph explored by the algorithm is an *ordered subgraph* G_0 of G, i.e. a subgraph s.t. whenever G_0 contains some edge e, then it also contains all its left brothers. A data graph containing G_0 as an ordered subgraph is called an *extension* of G_0.

The extension property: An edge $u \rightarrow v$ is in G_0 iff G_0 has an extension G' with some node v' s.t. $v' \in Q(G')$ and v' is either v, one of its right bothers, or a descendent of v or its right brothers.

Based on that we prove the following:

Theorem 4.2 *Let A be some other evaluation algorithm for the query Q,* [4] *Then for any data graph G, $cost(A, G) \geq cost(A_O, G)$.*

Proof: Assume there exists a data graph G for which A outperforms A_0. Due to the extension property, whenever A skips an edge in G that A_0 read, we can extend the subgraph G_0 seen so far to an instance G' where some answer of Q resides in a part that is not accessed by A (either below the unread edge, or below one of its right brothers). Since after skipping the edge, A cannot go back to read it or its right brothers (this is the computation model assumed), A, when running on G', does not compute a correct full answer. A contradiction. □

4.3 Transformations

Besides querying in a strict sense, query languages can be used to restructure and transform semistructured data or to integrate semistructured data from several sources. If we have validating schemas for both input and output data, a central issue then becomes whether the resulting data is valid w.r.t the given output schema. Of course, we can always check validity after query execution, but this may result in run time errors, which are hard to recover from. The *type checking* problem for transformation queries is: given schemas $S1$ and $S2$ and a query Q, check whether for any instance G of $S1$, $Q(G)$ conforms to $S2$. A related problem is *type inference*: given an input schema $S1$ and query Q, find the "most specific" schema $S2$ describing the possible query results.

We consider here transformations by means of Skolem functions. (An abstraction of the query languages for semistructured data described in the literature [PAGM96, BDHS96, FFK+98, CDSS97,

[3] Recall that, viewing the graph as an ADT with the above two traversal operations, the edges are traversed from left to right. Hence, to reach the b edge, these left brothers need to be traversed first.

[4] Recall that we focus here on a specific computation model and A is assumed to belong to the given class.

MZ98]). We show in the full paper that, in general, the type inference may not have a solution: there may not exist a schema best describing the transformation result. It exists however for transformations having one singled-variable Skolem function, and in the full paper we present an algorithm deriving it. The algorithm is based on traces, and its complexity is exponential time. In general one cannot do much better: we prove that the problem is PSPACE hard. But, as before, restricting the allowed schemas and queries leads to polynomial solution. The algorithm can be extended to handle transformations with arbitrary Skolem functions, but then its result is an approximation.

5 Conclusion

We studied in the paper the interaction between schemas and queries in semistructured data. We focused on type checking and inference, offering a nearly complete picture of the complexity of the problems, and introducing a novel *Traces* technique, proved to be useful for several other significant tasks such as query formulation, optimization, and validation. Type checking and type inference are well studied problems in functional programming languages [Mit90, Mit96]. The complexity there is derived from the interaction of function types, polymorphism, and let-bindings; in our setting the complexity is derived from the interaction of regular expressions in types with regular path expressions in queries. Optimization of simple queries with a single path expression, using types, has been considered in [FS98] and [BM99]. The setting we provide here is far more general.

References

[AV97] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 122–133, 1997.

[BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[BM99] Catriel Beeri and Tova Milo. Schemas for integration and translation of structured and semi-structured data. In *Proc. of the International Conference on Database Theory*, pages 296-313, Jerusalem, Israel, 1999. Springer Verlag.

[CDSS97] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 177-188, 1998.

[DFF+9u] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the International World Wide Web Conference*, 1999, to appear.

[DGM98] D.Calvanese, G.Giacomo, and M.Lenzerini. What can knowledge representation do for semi-structured data? In *Proc. of the 15th National Conf. on Artificial Intelligence(AAAI-98)*,1998.

[FFK+98] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[FS98] Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Data Engineering*, pages 14–23, 1998.

[Gol90] C. F. Goldfarb. *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proc. of Very Large Data Bases*, pages 436–445, Sept. 1997.

[Mit90] John C. Mitchell. Type systems for programming languages. In *Formal Models and Semantics*, volume B of

Handbook of Theoretical Computer Science, chapter 8, pages 367–458. Elsevier, Amsterdam, 1990.

[Mit96] John C. Mitchell. *Foundations for Programmng Languages*. MIT Press, 1996.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proc. of the Fourth Conf. on Parallel and Distributed Information Systems*, Miami, Florida, December 1996.

[MZ98] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proceedings of Very Large Data Bases*, pages 122-133, August, 1998.

[NAM97] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, 1997. Available from `http://www.research.att.com/~suciu/workshop-papers.html`.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of Very Large Data Bases*, pages 413–424, September 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE Int. Conference on Data Engineering*, pages 251–260, March 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructure heterogeneous information. In *International Conference on Deductive and Object Oriented Databases*, pages 319–344, 1995.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of 14th ACM SIGACT Symposium on the Theory of Computing*, pages 137–146, San Francisco, California, 1982.