Typing Semistructured Data

Gemo-Lamsade

January 23, 2008

In this chapter, we discuss the typing of semistructured data. We first present motivations and discuss the kind of typing that is needed. Semistructured data typing is often based on automata. We recall basic notion of automatas, first on words, then on ranked trees, finally on unranked trees, so XML. We consider the main two languages for describing XML types, DTD and XML schema. In a last section, we discuss the typing of graphs.

1 Motivations

Perhaps the main difference with typing in relational systems is that typing is not compulsory for XML. It is OK to have an XML document with no prescribed type. However, when developing and using software, types are essential. They provide a key ingredient for the interoperability between programs. Like dependencies for the relational model, they are also useful to protect data against improper updates. They present many other advantages, some briefly considered next.

Storage improvement Suppose that some XML document is very regular, say it contains a list of companies, with for each, an ID, a name, an address and the name of its CEO. This same information may be stored very compactly, for instance, without repeating the names of attributes such as address for each company. So, a priori knowledge of the type of the data may help improve its storage.

Performance improvement Consider the following XQuery query:

```
select X.title
from   Bib._ X
where  X.*.zip = "12345"
```

Knowing that the document consists of a list of *book*s and knowing the exact type of *book* elements, one may be able to rewrite the query to the query:

```
select X.title
from   Bib.book X
where  X.address.zip = "12345"
```

that is typically much cheaper to evaluate.

In this section, we consider that XML documents (at least some of them) are typed and that programs use the types. In particular, they verify the types. Often, such verification is dynamic. For instance, a Web server verifies the type when sending an XML document or when receiving it. Indeed,

XML data tend to be checked quite often because programs prefer to verify types dynamically (when they transfer data) than risking to run into data of unexpected structure during execution. It is also interesting, although more complicated, to perform static type checking, i.e. verify that a program receiving data of the proper input type only generates data of the proper output type.

More formally the probleme is as follows:

Input: an input type T_i and the code of function f
where f is Xquery, Xpath, XSLT, etc.

Output: Possibly some output type T_o

Verification: Is it true that $\forall d \models T_i, f(d) \models T_o$?

Inference: Find the smallest T_o such that $f(d) \models T_o$.

Consider for instance,

```
for $p in doc("parts.xml")//part[color="red"]
return <part><name>$p/name</name><desc>$p/desc</desc></part>
```

Assuming no constraint on the input type, it is easy to see that the result is of type:

```
(part ( name ( any ) desc ( any ) )*
```

The problem is rapidly undecidable because of "joins". Consider for instance the program:

```
for $X in Input, $Y in Input do { print (<b/>) }
```

Suppose that the input is:

```
<a/> <a/>
```

Then the result is

```
<b/> <b/> <b/> <b/>
```

In general, the result consists in i b-elements with $i = n^2$ for some $n \geq 0$. Such type cannot be described in XML schema. One can approximate it but not obtain a "best" result:

$$b^*$$
$$\varepsilon + b^2 b^*$$
$$\varepsilon + b^2 + b^4 b^*$$
$$\varepsilon + b^2 + b^4 + b^9 b^*$$
$$\ldots$$

2 Automata

XML was recently introduced. Fortunately, the model can benefit from a theory that is well established, automata theory. We briefly recall some standard definitions and results on automata over words. Then we mention without proof how they extend to ranked trees with some limitations. As previously mentioned, XML is based on unranked trees, so we consider finally unranked trees.

2.1 Automata on words

This is standard material. We recall here briefly some notation and terminology.

Definition: A *finite state automata* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where

1. Σ is a finite alphabet;
2. Q is a finite set of states;
3. $q_0 \in Q$ is the initial state;
4. $F \subseteq Q$ is the set of final states; and
5. δ, the transition function, is a mapping from $(\Sigma \cup \{\varepsilon\}) \times Q$ to 2^Q.

Such a nondeterministic automata accepts or rejects words in Σ^*. The set of words accepted by an automaton A is denoted $L(A)$. A language accepted by an FSA is called a *regular* language. They can alternatively be described as regular expressions built using concatenation, union and Kleene closure, e.g., $a(b+c)^*d$.

Example 2.1 Consider the FSA A with $\Sigma = \{a,b\}$, $Q = \{q_0,q_1,q_2,q_3\}$, $F = \{q_2\}$, $\delta(a,q_0) = \{q_0,q_1\}$, $\delta(b,q_1) = \{q_0\}$, $\delta(\varepsilon,q_1) = \{q_2\}$, $\delta(\varepsilon,q_2) = \{q_3\}$, $\delta(\varepsilon,q_3) = \{q_3\}$. Then *abaab* is not in $L(A)$ and *aba* is in $L(A)$.

An automaton is *deterministic* if (i) it has no ε-transition such as $\delta(\varepsilon,q) = ...$ and (ii) no alternative transition such as $\delta(a,q) = \{q', ...\}$.

The following important results are known about FSAs:

1. For each FSA A, one can construct an equivalent deterministic FSA B (i.e., a deterministic FSA accepting exactly the same words). In some cases, the number of states of B is exponential in that of A.
2. There is no FSA accepting the language $\{a^i b^i \mid i \geq 0\}$.
3. Regular languages are closed under complement.
   Just take $F' = Q - F$.
4. Regular languages are closed under union and intersection.
   Given two automata A, B, construct an automaton with states $Q \times Q'$ that simulates both. For instance, an accepting state for $L(A) \cap L(B)$ is a state (q, q'), where q is accepting for A and q' for B.

3 Automata on ranked trees

For words, there is absolutely no difference between left-to-right and right-to-left acceptance. For trees, there is a difference between top-down and bottom-up. Intuitively, in a top-down, we have a choice of the direction to go (e.g., to choose to go to the first child or the second) whereas in bottom-up like in automata for words, the direction is always prescribed.

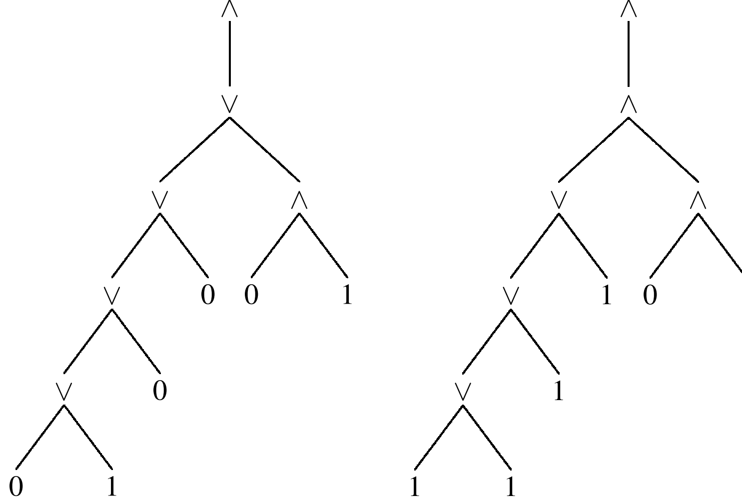

Figure 1: Two binary trees

Let us start with an example of bottom-up automata for binary trees. The transition function specifies a state for the leaf nodes. Then if $\delta(a,q,q')$ contains q''. This specifies that if the left and right children of a node labelled a are in states q,q', respectively, then the node *may move* to state q''.

Example 3.1 $\Sigma = \{0,1,\vee,\wedge\}$, $Q = \{f,t\}$, $F = \{t\}$, $\delta(0) = \{f\}$, $\delta(1) = \{t\}$, $\delta(\wedge,t,t) = \{t\}$, $\delta(\wedge,f,t) = \delta(\wedge,t,f) = \delta(\wedge,f,f) = \{f\}$, $\delta(\vee,f,f) = \{f\}$, $\delta(\vee,f,t) = \delta(\vee,t,f) = \delta(\vee,t,t) = \{t\}$. The tree of the left of Figure 1 is accepted by the bottom-up tree automata, whereas the one on the right is rejected.

An ε-transition in this context is of the form $\delta(a,r) = r'$, meaning that if a node of label a is in state r, then it may move to state r'. We can also define deterministic bottom-up tree automata by forbidding ε-transition and alternatives (i.e., some $\delta(a,q,q')$ containing more than one states.)

Definition: A set of trees is a *regular tree language* iff it is accepted by a bottom-up tree automata.

As for automata on words, one can "determinize" a non-deterministic automata. More precisely, given a bottom-up tree automata, one can construct a deterministic bottom-up tree automata that accepts the same trees.

Top-down tree automata In a top-down tree automaton, transitions are of the form $(q,q') \in \delta(a,q'')$ with the meaning that if a node labelled a is in state q'', then this transition moves its left child to state q and its right child to q'. The automaton accepts if all leaves are in accepting states. Determinism is defined in the obvious manner.

It is not difficult to show that a set of trees is regular iff it is accepted by a top-down automata.

Deterministic top-down automata are weaker. Consider the language $L = \{f(a,b), f(b,a)\}$. It is easy to see it is regular. Now one can verify that if there is a deterministic top-down automata accepting it, then it would also accept $f(a,a)$, a contradiction. Thus deterministic top-down automata are weaker.

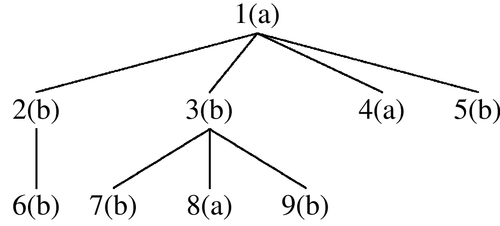

Figure 2: A tree with node identifiers listed

Generally speaking, one can show for regular tree languages the same results than for regular languages (sometimes the complexity is higher). In particular:

1. Given a tree automata, one can find an equivalent one that is deterministic (with possibly an exponential blow-up).

2. Regular tree languages are closed under complement, intersection and union (with similar proofs than for word automata).

3.1 Trees and monadic second-order logic

On can represent a tree as a logical structure using identifiers for nodes. For instance for the tree of Figure 2 is represented by:

$$E(1,2), E(1,3), ...E(3,9)$$
$$S(2,3), S(3,4), S(4,5), ..., S(8,9)$$
$$a(1), a(4), a(8)$$
$$b(2), b(3), b(5), b(6), b(7), b(9)$$

The syntax of *monadic second-order logic* is given by:

$$\varphi \quad :- \quad x = y \mid E(x,y) \mid S(x,y) \mid a(x) \mid ...\varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x\varphi X \mid \exists X\varphi \mid X(x)$$

where x is an atomic variable and X a variable denoting a set. Observe that in $\exists X\varphi$, we are quantifying over a set, the main distinction with first-order logic where quantification is only on atomic variables.

We can capture in MSO the constraint: "each a node has a b descendant". This is achieved by stating that for each node x labeled a, each set X containing x and closed under descendants contains some node labeled b. Formally,

$$\forall x(a(x) \rightarrow (\forall X(X(x) \wedge \beta(X) \rightarrow \exists y(X(y) \wedge b(y)))))$$
$$\text{where } \beta(X) = \forall y \forall z(X(y) \wedge E(y,z) \rightarrow X(z))$$

Now we have:

Theorem 3.2 A set L of trees is regular iff $L = \{T | T \models \varphi\}$ for some monadic second-order formula φ, i.e., if L is definable in MSO.

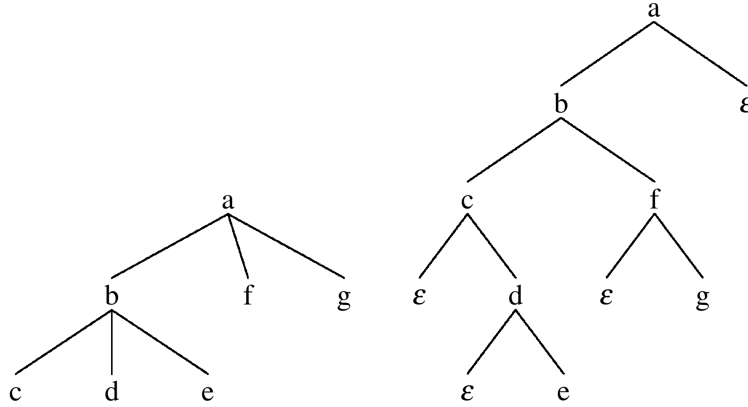

Figure 3: An unranked tree and its corresponding ranked one

3.2 Unranked trees

But XML documents are unranked as , for instance:

```
book(intro,section*,conclusion)
```

Reconsider the Boolean circuit example. Suppose we want to allow and/or gates with arbitrary many inputs. The set of transitions becomes infinite:

$$\delta(\wedge,t,t,t) = t, \delta(\wedge,t,t,t,t) = t ...$$
$$\delta(\wedge,f,t,t) = f, \delta(\wedge,t,f,t) = f ...$$
$$\delta(\vee,f,f,f) = f, \delta(\vee,f,f,f,f) = f ...$$
$$\delta(\vee,t,f,f) = t, \delta(\vee,f,t,f) = t ...$$

So an issue is to represent this infinte set of transitions. To do that, we can use regular expressions on words.

Example 3.3 Consider the following languages:

$$And1 = t+ \quad And0 = (t+f)^*f(t+f)^*$$
$$Or0 = f+ \quad Or1 = (t+f)^*t(t+f)^*$$

Then one can define infinite sets of transitions:

$$\delta(\wedge, And1) = \delta(\vee, Or1) = t, \qquad \delta(\wedge, And0) = \delta(\vee, Or0) = f$$

One can base a theory of unranked trees on that. Alternatively, one can build on ranked trees by representing any unranked tree by a binary tree where the left child represents the first child and the right child, the first sibling. See Figure 3.

Let F the bijection that encodes an unranked tree T into $F(T)$, the binary tree with first-child and first-sibling. Let F^{-1} be the inverse mapping that "decodes" binary trees thereby encoded. One can

show that for each unranked tree automata A, there exists a ranked tree automata accepting $F(L(A))$. Conversely, for each ranked tree automata A, there is an unranked tree automata accepting $F^{1}(L(A))$. Both constructions are easy.

As a consequence, one can see that unranked tree automata are closed under union, intersection and complement.

Determinism is defined as follows:

$\forall a \in \Sigma, \forall w \in Q^*$, there exists a unique rule $\delta(a, L)$ such that $w \in L$.

One can also "determinize" unranked tree automata. (It does not suffice to "go through" ranked tree automata because the translation from ranked to unranked does not preserver determinism),

4 Document Type Definition

DTD stands for "Document Type Definition". This is the oldest and still very much used syntax for specifying typing constraints on XML. To describe types for XML, the main idea is to describe the children that nodes with a certain label may have. With DTDs, the labels of children of a node of a given label are described by regular expressions: The syntax is bizarre but rather intuitive:

An example of a DTD is as follows:

```
<!ELEMENT populationdata (continent*) >
<!ELEMENT continent (name, country*) >
<!ELEMENT country (name, province*)>
<!ELEMENT province (name, city*) >
<!ELEMENT city (name, pop) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT pop (#PCDATA) >
```

The "*" indicate an arbitrary number of children with that structure; PCDATA just means string. In DTDs, the regular expressions are supposed to be *deterministic*. In brief, the XML data can be parsed (and its type verified) by a deterministic finite state automata For instance, the expression

$$(a+b)^*a$$

is not deterministic since when parsing a first a, one doesn't know whether this is the a of $(a+b)*$ or that of the last a. On the other hand, this expression is equivalent to

$$(b^*a)(b^*a)^*$$

that is deterministic

Under this restriction, it is easy to type check some XML data while scanning it, e.g., with a SAX parser. Observe that such a parsing can be sometimes performed with a finite state automata but that sometimes more is required. For instance, consider the following DTD:

```
<!ELEMENT part ( part* ) >
```

The parser reads a list of *part* opening tags. A stack (or a counter) is needed to remember how many where found to verify that the same number of closing tags is found.

We do want the kind of recursive definitions that cannot be verified simply by an FSA. On the other hand, DTDs present features that are less desired, most importantly, they are not closed under union:

```
DTD1:   <!ELEMENT used ( ad*) >
        <!ELEMENT ad ( year, brand )>
DTD2:   <!ELEMENT new ( ad*) >
        <!ELEMENT ad ( brand )>
```

$L(DTD1) \cup L(DTD2)$ cannot be described by a DTD although it can be described easily with a tree automata. It turns out that DTDs are also not closed under complement. The issue here is that the type of *ad* that depends of its parent. We can approximate what we want:

```
<!ELEMENT ad (year?, brand) >
```

But this is only an approximation.

What we need to do is to decouple the notions of type and that of label. Each type corresponds to a label, but not conversely So, for instance, we may want to types for ads, with the same label:

```
car:  [car]  ( used + new )*
used: [used] (ad1*)
new:  [new]  (ad2*)
ad1:  [ad]   (year, brand)
ad2:  [ad]   (brand)
```

With such decoupling, we can prove closure properties. This is leading to XML schemas that are based on decoupled tags with many other “gadgets”.

5 XML schema

XML schema has often been criticized for being unnecessarily complicated. This syntax is proposed by W3C and boosted by industry because it is used notably in Web services. XML schemas are very close to deterministic top-down tree automata but as already mentioned with many practical gadgets. It is using an XML syntax, so it benefits from XML tools such as editors and type checkers. So, in particular, the type of all XML schema definitions can be described with an XML schema.

An example of XML schema is:

```
<?xml version="1.0" encoding="utf-8"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             targetnamespace="http://www.net-language.com">
    <xs:element name="book">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="title" type="xs:string"/>
          <xs:element name="author" type="xs:string"/>
          <xs:element name="character"
                      minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="friend-of" type="xs:string"
                            minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="since" type="xs:date"/>
```

```
          <xs:element name="qualification" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="isbn" type="xs:string"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

XML schemas first include the definition of simple elements with atomic types, where the common types are xs:string, xs:decimal, xs:integer, xs:boolean, xs:date, xs:time. For instance, one cand define:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

And corresponding data are:

```
<lastname>Refsnes</lastname>
<age>34</age>
<dateborn>1968-03-27</dateborn>
```

One can also define attributes as, for instance in:

```
<xs:attribute name="lang" type="xs:string"/>
```

with for corresponding data

```
<lastname lang="EN">Smith</lastname>
```

Then we can have complex elements as in:

```
<product pid="1345"/>
<employee><firstname>John</firstname><lastname>Smith</lastname></employee>
<food type="dessert">Ice cream</food>
<desc>It happened on <date lang="norwegian">03.03.99</date></desc>
```

A complex element can be empty, containt text, other elements or be “hybrid”, i.e. contain both some text and subelements.

One can impose restrictions of simple elements as in:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/><xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Other restrictions are: enumerated types, patterns, etc.

One can impose restriction on complex elements as in:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

It is possible to specify a type and give it a name.

```
<xs:complexType name="personinfo">
  <xs:sequence> <xs:element name="firstname" type="xs:string"/>
  <xs:element name="lastname" type="xs:string"/> </xs:sequence>
</xs:complexType>
```

Then we can use this type name in a type declaration, as in:

```
<xs:element name="employee" type="personinfo" />
```

One should also mention some useful gadgets

1. It is possible to import types associated to a namespace

```
<import nameSpace = "http:// ..."
   schemaLocation = "http:// ..." />
```

2. It is possible to include an existing schema

```
<include schemaLocation="http:// ..."/>
```

3. It is possible to extend/redefine an existing schema

```
<redefine schemaLocation="http:// ..."/>
   ....   Extensions ...
   </redefine>
```

To conclude, consider in Figure 4. The corresponding XML schema is given in Figure 5.

There are more restrictions on XML schemas, some rather complex ones. For instance, inside an element, no two types may use the same tag. Some of them can be motivated by the requirement to have efficient type validation. The main difference with DTDs (besides some useful gadgets) is that the notions of types and tags are decoupled.

```
<!ELEMENT EMAIL (TO+, FROM, CC*, BCC*, SUBJECT?, BODY?)>
<!ATTLIST EMAIL
  LANGUAGE (Western|Greek|Latin|Universal) "Western"
  ENCRYPTED CDATA #IMPLIED
  PRIORITY (NORMAL|LOW|HIGH) "NORMAL">
<!ELEMENT TO (#PCDATA)>
<!ELEMENT FROM (#PCDATA)>
<!ELEMENT CC (#PCDATA)>
<!ELEMENT BCC (#PCDATA)>
<!ATTLIST BCC
  HIDDEN CDATA #FIXED "TRUE">
<!ELEMENT SUBJECT (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ENTITY SIGNATURE "Bill">
```

Figure 4: DTD

6 Graphs and bisimulation

6.1 Graph semistructured data

With ID-IDREF, XML data are moving from trees to graphs. Different models have been proposed for describing graphs. In some sense, RDF is also a graph data model. We briefly formalize next a graph data model.

Definition: (Object Exchange Model) An OEM is a finite, labeled, rooted graph (N,E,r) (simply (E,r) when N is understood) where:

1. N is a set of nodes,
2. E is finite ternary relation subset of $N \times N \times \mathscr{L}$ for some set $\mathscr{L}$ of labels,
   $E(s,t,l)$ indicates there is an edge from s to t labeled l,
3. r is a node in the graph.

For trees, we sometimes ignore the ordering of children of a node and possibly see them as a set (i.e., ignore repetitions as well). So we may ignore the difference between $a(1,2,2,1,5)$ and a(1,2,5). This is somewhat suggesting a similar choice for graphs that is the basis of bisimulation and some typing scheme for graph data.

6.2 Graph bisimulation

A *simulation* S of (E,r) with (E',r') is a relation between the nodes of E and E' such that $S(r,r')$ and if $S(s,s')$ and $E(s,t,l)$ for some s,s',t,l, then there exists t' with $S(t,t')$ and $E'(s',t',l')$.

The intuition is that we can simulate moves in E by moves in E'.

Given $(E,r),(E',r')$, S is a *bisimulation* if S is a simulation of E with E' and S^{-1} is a simulation of E' with E.

One may wonder what this have to do with typing? Take a very complex graph E. We can describe it with a “smaller” graph E' that is a bisimulation of E. There may be several bisimulations

```
<?xml version="1.0" ?>
<Schema name="email" xmlns="urn:schemas-microsoft-com:xml-data"
                     xmlns:dt="urn:schemas-microsoft-com:datatypes">
 <AttributeType name="language"
                dt:type="enumeration" dt:values="Western Greek Latin Universal" />
 <AttributeType name="encrypted" />
 <AttributeType name="priority" dt:type="enumeration" dt:values="NORMAL LOW HIGH" />
 <AttributeType name="hidden" default="true" />
 <ElementType name="to" content="textOnly" />
 <ElementType name="from" content="textOnly" />
 <ElementType name="cc" content="textOnly" />
 <ElementType name="bcc" content="mixed">
 <attribute type="hidden" required="yes" />
 </ElementType>
 <ElementType name="subject" content="textOnly" />
 <ElementType name="body" content="textOnly" />
 <ElementType name="email" content="eltOnly">
 <attribute type="language" default="Western" />
 <attribute type="encrypted" />
 <attribute type="priority" default="NORMAL" />
 <element type="to" minOccurs="1" maxOccurs="*" />
 <element type="from" minOccurs="1" maxOccurs="1" />
 <element type="cc" minOccurs="0" maxOccurs="*" />
 <element type="bcc" minOccurs="0" maxOccurs="*" />
 <element type="subject" minOccurs="0" maxOccurs="1" />
 <element type="body" minOccurs="0" maxOccurs="1" />
 </ElementType>
 </Schema>
```

Figure 5: XML schema

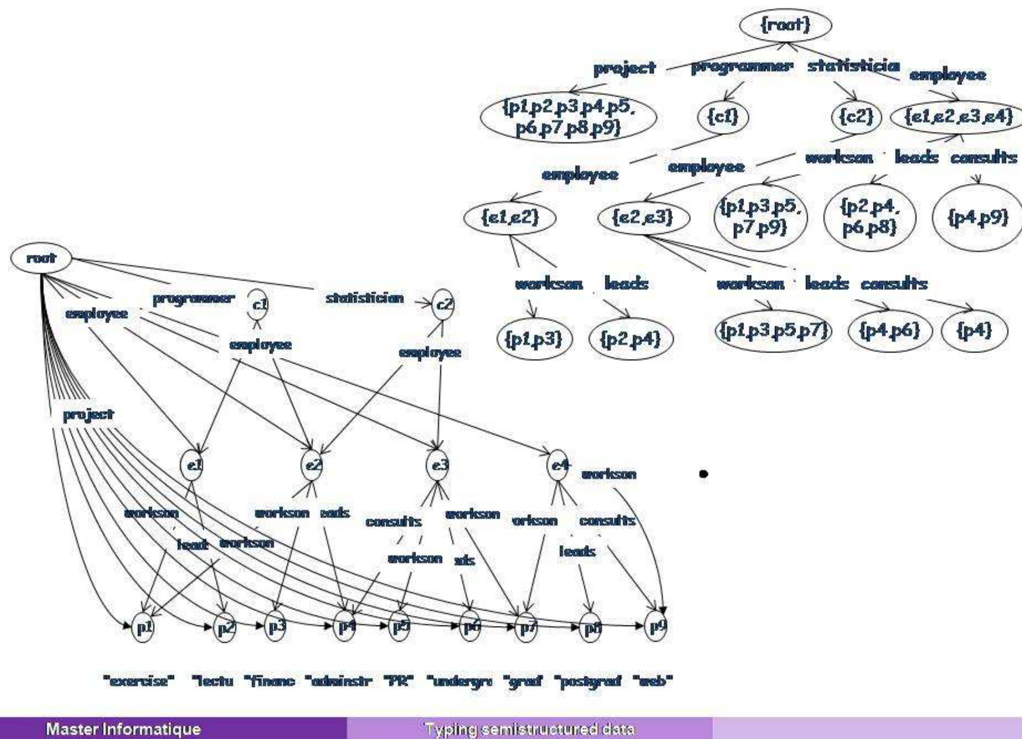

Figure 6: Data guide

for E including more and more details. At one extreme, we have the graph consisting of a single node with a self loop. At the other extreme, we have the graph E itself.

6.3 Data guides

Sometimes we are interested only in the paths from the root. This may be useful for instance to provide an interface that allows to navigate in the graph. Consider the OEM Graph of Figure 6. There are paths such as

```
programmer
programmer employee
programmer employee workson/
```

It turns out that this set of paths is regular. A deterministic automaton accepting it is called a *data guide*. Observe that the data guide gives information about the structure of the graph, but only a limited one: for instance, it does not distinguish between:

```
a ( b ( c d ) )
a ( b ( c ) b ( d ) )
```