Techniques and Rule Patterns for Declaratively Querying Web Data with FLORID

Bertram Ludäscher Rainer Himmeröder Wolfgang May
Institut für Informatik, Universität Freiburg, Germany
{ludaesch,himmeroe,may}@informatik.uni-freiburg.de

Abstract

FLORID is an implementation of the deductive object-oriented database language F-logic and has recently been extended to provide a declarative semantics for querying the Web. By means of several illustrative examples, we show how FLORID's rule-based logical language can be used to extract, query, and analyze data from the Web.

1 Introduction

Models and languages for querying the Web, for handling semistructured data, and for integration and restructuring of information have recently attracted a lot of interest [MV98]. We argue that *dood* languages, i.e., supporting deductive and object-oriented features, are particularly suited in this context: *Object-orientation* provides a flexible common data model for combining information from heterogeneous sources and for handling partial information. Techniques for navigating in object-oriented databases can be applied to semistructured databases as well, since the latter may be viewed as (very simple) instances of the former. *Deductive rules* provide a powerful framework for expressing complex queries in a high-level, declarative programming style. WebLog [LSS96] and ADOOD [GMNP97], for example, build upon the dood language F-logic [KLW95]. FLORID [FLO] is an implementation of F-logic, which has been extended to provide a declarative semantics for querying the Web. The proposed extension allows extraction and restructuring of data from the Web and a seamless integration with local data. A main advantage of the approach is that it brings together the above-mentioned issues in a unified, formal framework and supports rapid prototyping and experimenting with all these features. In particular, FLORID programs may be used (simultaneously) as wrappers, mediators, or for "ordinary" deductive queries.

In this paper we illustrate, by means of several examples, the different techniques and rule patterns for declaratively querying the Web with FLORID. The examples substantiate the claim that a dood framework is suited for querying and management of semistructured and/or Web data. In Section 2, we briefly introduce the basics of F-logic, FLORID's Web model, and the data-driven style of accessing Web documents. Section 3 deals with *structure-based* queries, i.e., involving only the hyperlink structure of Web documents. In contrast, Section 4 focuses on *content-based* queries, i.e., where also the textual representation of documents has to be analysed in order to extract data (in general, FLORID programs involve both structure and contents of Web documents). Some concluding remarks and further FLORID references are given in Section 5.

2 Exploring the Web with FLORID

We briefly review the basic constructs of F-logic and its extension by path expressions.

A Glimpse of F-Logic. Consider the following fragment of an F-logic program:

```
person[name ⇒string; children@(integer) ⇒⇒person].          % signature of class person
employee::person.                                            % subclass relationship
john:employee[                                               % instance relationship and
    name →"John Smith"; children@(1998) →→{mary,bob}].       % ... some example data
X.father:man ←X:person.                                      % object creation by ...
X.mother:woman ←X:person.                                    % ... path expressions
```

First, the *signature* of class person is specified: The *single-valued* method name yields instances of class string, whereas the *multi-valued* (and parameterized) method children yields instances of person. The *subclass* relation employee::person states that all members of employee are also members of person. Next, john:employee defines that the object named john is an *instance* of class employee; the specification inside [...] defines the actual data values for name and children.

Path Expressions and Object Creation. The last two rules demonstrate how path expressions in the head can be used to create new object identifiers (oids): If X is bound to an instance of person, then the single-valued method father becomes defined for X. The newly "created" father is referenced by the *path expression* X.father and is made an instance of man. In particular, john[father→john.father] and (john.father):man hold (similarly for mother and woman). Thus, the dot "." corresponds to navigation along single-valued methods (→) like name and father, while ".." is used to navigate along multi-valued methods (→→) like children, e.g., as in john..children@(Y)[surname→john.surname].

The use of path expressions for object creation is crucial for our approach to data-driven exploration of the Web using F-logic. Object creation has to be used with care in order to avoid infinite universes and nontermination: e.g., if a rule X.father:man ← X:man were added to the program, an infinite number of objects like john.father, john.father.father, etc. would be created.

2.1 The Web Model

As usual, we conceive the Web as a graph-like structure consisting of documents and links between them. More precisely, we distinguish between the class url of (potential) urls, and webdoc of (accessed) Web documents (Fig. 1). Urls are instances of string, for which a special method get is definable (see below). Typical elements of class webdoc are HTML pages, but other document types may also be included (e.g., BibTEX). In particular, one may define a subclass sgmldoc such that the parse-tree of fetched SGML documents can be analyzed:[1] If a Web document has been accessed, a number of *system-defined* methods may become defined for it: e.g., url (the url of the Web document), modif (time of last modification), type, and—most notably—the multi-valued method hrefs@(*label*) representing the outgoing links of the Web document (see Fig. 1). Note that hrefs is parameterized with the label of the link. If the Web access fails, error returns the reason of the failure (e.g., *page not found*).

[1] This feature is currently being incorporated into FLORID.

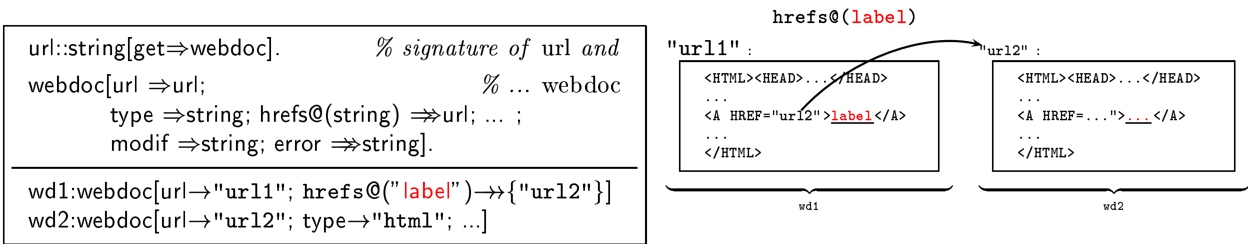

Figure 1: F-logic Web model: signature and example data

2.2 Data-Driven Web Exploration

A Web document is accessed and added to the local F-logic database by defining the method get for an instance u of class url in the head of a rule, thereby creating the new oid "u.get" of the fetched Web document. After the oid u.get has been created, system-defined methods are automatically "filled in" by FLORID, and the Web document named u.get becomes an ordinary F-logic object (conceivable as a large string). Thus, u.get is "cached" and the url u is accessed only once. Note that the *potentially* system-defined methods for class webdoc are given by the FLORID system—the *actually* system-defined (i.e., "filled") methods for u.get depend on the result of accessing the url u. For example, if error is defined, then hrefs@(*label*) will be undefined. Here, the advantages of using an object-oriented framework like F-logic become apparent: although the instances of a certain class may typically define a certain set of methods, some (or all) of these methods may be omitted. Thus, the use of NULL values as in the relational model can be avoided.

Apart from this first analysis of the document done by the system, the main power of the approach lies in the possibility to specify arbitrary *user-defined* methods for fetched documents using all features of F-logic (and path expressions). Consider, e.g., the following F-logic program:

```
"http://www.informatik.uni-freiburg.de/~dbis/":url.get.    % (1) define and access start url
U:explored ← U:url.get.                                     % (2) remember explored documents
U1.get ← U:explored.get[hrefs@(Lbl) →→{U1}],                % (3) transitively access url U1 ...
             substr(U,U1).                                   % ... if U is a prefix of U1
```

The path expression in (1) defines a particular string as an instance of url and, since the method get is defined for this url, accesses the corresponding document. Due to (2), the urls of all accessed documents become instances of class explored. (3) uses the power of deduction with recursion and transitively accesses all documents reachable from an already explored url U, provided that the referenced url U1 is a substring of U. Thus, only urls starting with http://... ~dbis will be accessed; for other links, get remains undefined. Therefore, explored and unexplored urls can be distinguished using the queries ?-U:explored and ?-U:url, not U:explored, respectively. In particular, only a finite number of new oids u.get is created.

Since exploration of the Web is completely data-driven, a seamless integration with the bottom-up evaluation strategy of FLORID and a declarative semantics of the language is achieved [HLLS97, LHL+98].

3 Querying Structure

When considering Web documents, one may distinguish between *structure* (i.e., how pages are interlinked) and *contents* (i.e., the actual data provided on pages; see Section 4). The hyperlink structure of a collection of Web pages (cf. Fig. 2) can be explored in FLORID by navigating along the multi-valued method hrefs@(*label*). Indeed, the link structure of a set of Web documents is the prototypical example of what is called a *semistructured database* (*ssdb*), but also the data found on individual pages can often be considered as semistructured.

3.1 Semistructured Databases

The enormous success of the Web has recently lead to an increasing interest in models and languages for *ssd* [Abi97, AQM+97, BDHS96, Suc97]. Typical features attributed to *ssd* include the following: the structure is irregular, partial, unknown, or implicit in the data, and typing is not strict but only indicative [Abi97]. Since the distinction between schema and data is often blurred, semistructured data is sometimes called "self-describing" [Bun97].

In general, there may be very little structure in semistructured data, or the structure may be contained within the data and has to be discovered. Therefore, the underlying data model has to be simple and flexible. Here, we adopt the fairly standard model where *ssd* is represented as a labeled graph:

Definition 1 Let $\mathcal{N}$ be a set of *nodes* and $\mathcal{L}$ a set of *labels*. A *semistructured database*[2] (*ssdb*) $\mathcal{D}$ is a finite subset of the set of *labeled edges* $\mathcal{E} = \mathcal{N} \times \mathcal{L} \times \mathcal{N}$. □

Like in other object-oriented data models, data in an F-logic database instance can be conceived as a labeled graph: e.g., the F-logic atom $X[M@(A_1, \ldots, A_k) \twoheadrightarrow \{Y\}]$ corresponds to an edge from object X to Y, which is labeled with the parameterized method $M@(A_1, \ldots, A_k)$. The *ssd* model is a special case of this graphical representation:

Nodes of a *ssdb* $\mathcal{D}$ correspond to oids, whereas labeled edges correspond to multi-valued method applications. More precisely, for a labeled edge in $\mathcal{D}$, an equivalent graph notation and a representation in F-logic syntax can be given as follows:

$$(x, \ell, y) \in \mathcal{D} \quad \Leftrightarrow \quad x \overset{\ell}{\twoheadrightarrow} y \quad \Leftrightarrow \quad x[\ell \twoheadrightarrow \{y\}] \ .$$

Thus, for a given *ssdb* $\mathcal{D}$, we obtain the following natural representation in F-logic:

$$\mathsf{X : node,\ Y : node,\ L : label,\ X[L} \twoheadrightarrow \mathsf{\{Y\}]} \leftarrow \mathsf{ssdb(X,L,Y).}$$

Here, it is essential that L is viewed as a *multi-valued* method, since there may be several distinct edges emanating from x which share the same label ℓ.

Web Skeleton. Clearly, the link structure of a set of Web document may be conceived as a *ssdb*: nodes correspond to Web documents and edges to labeled hyperlinks between documents. If nodes are opaque, i.e., when no information apart from the link structure is available, we speak of the *(Web) skeleton* of a set of Web documents. For example, the labeled graph depicted in Figure 2 represents a fragment of the skeleton of the DBLP server [DBL]: In the skeleton view, the only information available is contained in labels (represented as strings), whereas nodes are opaque. Thus, the *skeleton* covers the *structural* aspect of Web documents but not their *contents*. We will deal with extracting contents in Section 4.

[2] Sometimes also called *graph database* [BDFS97].

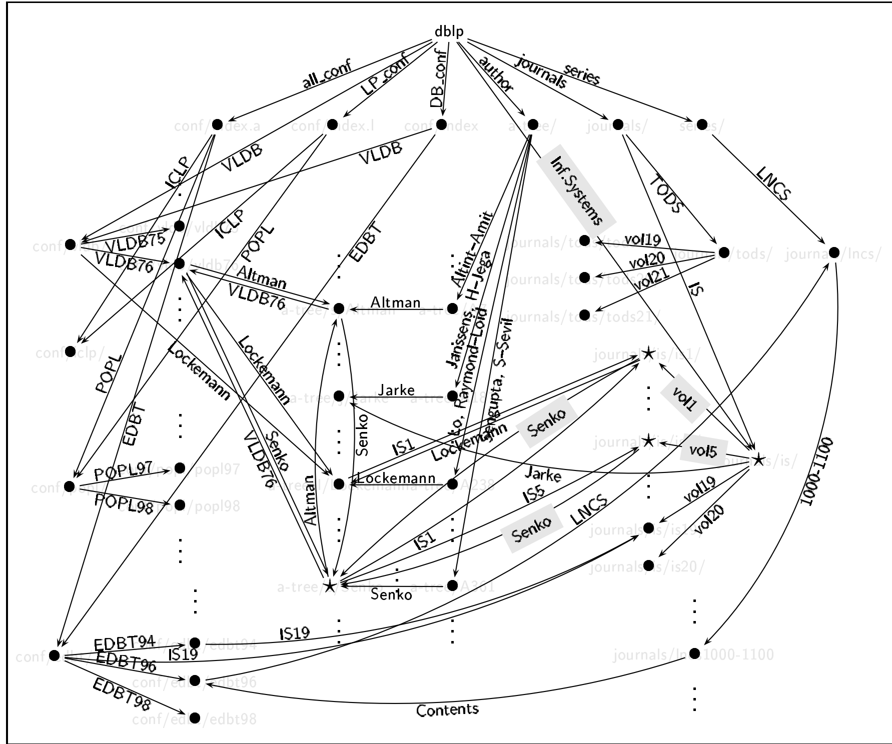

Figure 2: Fragment of the skeleton of the DBLP database

3.2 A Generic Web Skeleton Extractor

A Web skeleton like the one depicted in Fig. 2 can be automatically extracted with FLORID using the following generic skeleton extractor P_{ext}:

P_{ext}:

root[src$\twoheadrightarrow\{u_1, \ldots, u_n\}$].	% (1)
node :: url.	% (2)
U : node.get $\leftarrow$ root[src$\twoheadrightarrow\{$U$\}$].	% (3)
Y : node, L : label, X[L$\twoheadrightarrow\{$Y$\}$] $\leftarrow$ X : node.get[hrefs@(L)$\twoheadrightarrow\{$Y$\}$], φ.	% (4)
Y.get $\leftarrow$ Y : node, ψ.	% (5)

Figure 3: A generic skeleton extractor for FLORID

First, the relevant source url's are defined (1), and the class node is declared a subclass of url (2). Every source url u is made an instance of node (and thus of url), and the single-valued method get is defined for u (3): u is accessed, u.get is assigned the respective Web document and some additional methods for u.get are defined, among them hrefs@(...). Then, an exploration cycle is started: Given an url (node) x, in rule (4), for all labels ℓ and referenced url's y s.t. x.get[hrefs@(ℓ)$\twoheadrightarrow\{y\}$], the labeled edge $x \overset{\ell}{\twoheadrightarrow} y$ is added to the F-logic *ssdb* and y is made a node. By constraining (4) with an additional goal φ, only those labeled edges are defined which are considered relevant. Finally, (5) fetches the new Web document y.get of

the node y, provided the condition ψ holds. Thus, φ is a first constraint limiting the number of strings which are considered as nodes (and thus url's). Additionally, only those url's y are actually accessed and their contents retrieved in y.get, for which ψ holds.

The rule pattern given by P_{ext} allows straightforward extraction of a Web skeleton, simply by instantiating the source url's, and φ and ψ appropriately:

Example 1 (DBLP Skeleton Extractor) A fragment of the DBLP server skeleton (Fig. 2) is retrieved when starting the skeleton extractor with

```
root[src->>{dblp}].          dblp = "http://www.informatik.uni-trier.de/~ley/db/".
```

and the constraints

- φ = substr("trier",Y), and (*consider only url's containing "trier"*)
- ψ = substr("/db/journals/is/",Y). (*restrict to* is *(=Information Systems) journal*)

After all relevant documents have been accessed, we may query the skeleton: For example, we may be interested in all authors whose name contains a certain substring (say: "sen"), and who had a paper in *Information Systems*. Using the extracted hyperlink structure of the DBLP skeleton (cf. Fig. 2), we can directly issue the following query:

```
?- dblp.."Inf. Systems"..V..A, substr("sen",A).
   A/"Christian S. Jensen" V/"Volume 21, 1996"
   A/"Christian S. Jensen" V/"Volume 19, 1994"
   A/"Arun Sen" V/"Volume 20, 1995"
   A/"Arun Sen" V/"Volume 11, 1986"
   A/"Georg Lausen" V/"Volume 8, 1983"
      ⋮
   A/"Michael E. Senko" V/"Volume 1, 1975"
16 output(s) printed
```

As this query reveals, the links emanating from the *Information Systems* page are (typically) volume pages; the links emanating from volume pages are (typically) author pages. In this way, the link structure of Web sites can be explored and analysed using FLORID's powerful ad-hoc queries. □

Simply by instantiating P_{ext} differently, we obtain the skeleton of the weather encyclopedia of the *Schweizer Fernsehen (SF DRS)*:

Example 2 (Meteo Skeleton Extractor) We start P_{ext} as follows:

```
root[src->>{meteo}].     meteo = "http://www.sfdrs.ch/sendungen/meteo/lexikon/index.html".
```

and use the identical constraints

- φ, ψ = substr("/lexikon/",Y). (*restrict to "lexikon" pages*)

In order to find the names of all entries in the weather encyclopedia, we ask:[3]

[3] FLORID has different output modes for displaying answers: In Example 1 this mode is set to *variable bindings* (as in Prolog), while it is set to *ground instances* here (and in other examples below).

```
?– meteo..L.
  meteo.."absolute Feuchte".
  meteo.."Absorption".
    ⋮
  meteo.."Weiterführende Literatur".
131 output(s) printed
```
□

It should be clear that, apart from the above substr-predicate, there are many other ways of limiting the set of explored url's in P_{ext}. For example, one may define a method depth for each node, such that source nodes have depth zero and a document referenced from another one with depth n has itself depth at most $n+1$. Based on this, one can easily restrict Web exploration to documents with depth $\leq n_0$.

3.3 Mining Links

Once a Web skeleton has been extracted, FLORID's deductive query language allows to gather new information from the given data sources. We illustrate some features based on the weather encyclopedia program from Example 2. Operating on the skeleton of such an encyclopedia is particularly interesting, since its hyperlink structure mirrors—to a large extent—semantic relationships between the corresponding notions of the encyclopedia. For example, one may discover related notions using path expressions, or one may estimate the importance of certain notions based on the frequency of their use (which requires the use of aggregation):

Path Expressions. Take some entry from the weather encyclopedia, say "Ozonschicht" (*ozone layer*). Then, the query

```
?– meteo..L.."Ozonschicht"..M.
```

finds all entries L and M "in the context" of this entry, i.e., starting from the root page meteo, there is a link labeled L leading to the "Ozonschicht"-page, and from this page there is a link labeled M:

```
?– meteo..L.."Ozonschicht"..M.
  meteo.."Absorption".."Ozonschicht".."Atmosphäre".
  meteo.."FCKW".."Ozonschicht".."Atmosphäre".
    ⋮
  meteo.."O 3".."Ozonschicht".."FCKW".
  meteo.."UV-Strahlung".."Ozonschicht".."absorbiert".
49 output(s) printed
```

If we were only interested in "back-loops", i.e., labels which occur before *and* after the "Ozonschicht"-page, we could require that L=M, resulting in only four answers for L.[4]

General Path Expressions. Apart from the (simple) path expressions considered so far, so-called *general path expressions* have been suggested for querying and navigating on semistructured data (see e.g. [AQM+97]). Given labels L and M, these expressions allow to follow "generalized" labels like $(L{\cdot}M)$ (sequence), $(L|M)$ (disjunction), $(L)^*$ (iteration), $(L)^{-1}$ (inversion), etc. In [LHL+98] it is shown how such general path expressions can be evaluated with FLORID.

[4] "FCKW", "Ozon", "Stratosphäre", and "UV-Strahlung"

Aggregation. To find the "most important" entries, we can use aggregation and count the *fan-in*, *fan-out*, and *edge-count* of nodes and labels, respectively. This is accomplished by adding the following rules to the Meteo skeleton extractor:

```
e(X,L,Y)[] ← X:node[(L:label)→→{Y:node}].          % define all "edge-objects"
X[fan_out→N] ← N=count{Y [X]; e(X,_,Y)[]}.   % aggregation: count outgoing...
Y[fan_in→N] ← N=count{X [Y]; e(X,_,Y)[]}.              % ... and incoming edges
L[edge_count→N] ← N=count{Z [L]; Z=e(X,L,Y)[]}.      % count edges per label
```

The first rule defines, for each labeled edge $x \overset{\ell}{\twoheadrightarrow} y$, a distinct object (note the use of "[]" to distinguish the *object* e(X,L,Y)[] with empty specification from the ternary *predicate* e(X,L,Y)). The second and third rule use the built-in aggregation count: the expressions with curly braces define the aggregation: for example, "count{Y [X] ; ...} " means *"count all Y's, group by X"*, the expression behind the semicolon ";" is the aggregation goal.

In our example, when querying the edge-count, we find that "Luft" and "Temperatur" are the "most important" entries:

```
?- N = L.edge_count.
  1 = "absolute Feuchte".edge_count.
    ⋮
  35 = "Temperatur".edge_count.
  41 = "Luft".edge_count.
252 output(s) printed
```

Graph Algorithms. Since the Web skeleton is a directed labeled graph, general graph algorithms can be used to reveal interesting structural properties of the skeleton. For example, we may be interested in the *strongly connected components* (*scc*) of the given skeleton: two nodes x and y belong to the same *scc* iff they are reachable from each other in the given graph. To this end, we simply add the generic program for computing *scc*'s on *ssdb*'s (Figure 4) to the program above:

```
tc(X,Y) ← X:node[(_:label)→→{Y}].                       % on the given labeled graph...
tc(X,Y) ← tc(X,Z), tc(Z,Y).                       % ... compute the transitive closure

X:scc_id(X) ← X:node.                        % initially, each node belongs to his own scc

scc_id(X) = scc_id(Y) ← tc(X,Y), tc(Y,X).   % ...but scc's may be fused if mutually reachable!

?- sys.strat.doIt.                               % sys-command to enforce stratification

Z[size→N] ← N = count{X [Z] ; X:Z, Z = scc_id(C)}.       % determine the size of each scc
```

Figure 4: P_{scc}: Computing strongly connected components in FLORID

The program P_{scc} makes essential use of *derived equalities*—a special feature of F-logic (and FLORID). Also, note the user-defined stratification (?-sys.strat.doIt.) before the last rule: In order to ensure that the final aggregation produces the intended results, the rules above the aggregation must have been evaluated completely beforehand.
Given the rules above, we can now determine the number of *scc*'s and their sizes:[5]

[5]Here we have switched back to FLORID's *variable bindings* output mode.

```
?- M = count{Z [N] ; Z[size →N]}.
   M/1    N/96
   M/25    N/1
2 output(s) printed
```

Thus, there are 26 *scc*'s: 1 large *scc* with 96 nodes and 25 trivial *scc*'s with a single node. Observe that the argument U of the oid scc_id(U) is an instance of node and hence of url.[6] Therefore, we may apply the method get to it. In this way, we can extract the *titles* of pages in the large *scc* as follows:

```
?- scc_id(_U)[size→96], _U.get[title→T].
   T/"Wetterlexikon: Index"
   T/"Wetterlexikon: Absolute Feuchte"
      ⋮
   T/"Wetterlexikon: Wolken"
96 output(s) printed
```

4 Querying Contents

Apart from extracting the skeleton of a set of Web documents (i.e., their *link structure*), also their *contents* may be queried. To this end, built-in predicates for extracting and analyzing data from accessed Web documents have to be provided. A simple, yet flexible and powerful approach used in FLORID, is to view Web documents as (large) strings and then apply *regular expressions* to extract data. Regular expression can be used, for example, to extract all strings between pairs of HTML tags like <h2> and </h2> (level-2 headings), or to analyze tables or lists.[7] The regular expressions employed in FLORID include groups and format strings, thereby providing a very expressive language: The predicate

pmatch(*Str*, *RegEx*, *Fmt*, *Res*)

finds all strings in the input string *Str* which match the pattern given by the (Perl) regular expression *RegEx*. The *format string Fmt* describes how the matched strings should be returned in *Res*. This feature is particularly useful when using *groups* (expressions enclosed in (...)) in regular expressions. For example, we have:[8]

```
?- pmatch("A man's only as old as he feels", "/(.*) (he .*)/", "$1 the woman $2", X).
   X/"A man's only as old as the woman he feels"
1 output(s) printed
```

Since for an url u, the reference u.get denotes the fetched Web document, u.get can be used as first argument to the pmatch predicate.

4.1 Syntactical Queries: CIA World Factbook

We illustrate simple content-based queries using the pages of the *CIA World Factbook*, a collection of Web pages providing information about the countries in the World (e.g., on geography, people, government, and economy). Although one may argue that the data in the World Factbook is highly regular and should be put on the Web as a database in the

[6] See (2) in Figure 3.

[7] Another possibility currently being incorporated is to use a general SGML parser, whose output is directly mapped into some F-logic structure.

[8] The answer for X is a quote from Groucho Marx, see http://www.bmacleod.com/groucho.html

first place, there are some idiosyncrasies and irregularities to consider (see below for a simple example, i.e., the "pseudo-values" of the attribute capital). Moreover, the actual database is *not* available from the Web, whereas the semistructured HTML pages are.

Note that the following rules make up a *self-contained* program, i.e., there are no separate languages for wrappers or mediators. The program is roughly organized as follows: After a certain root page has been accessed, several outgoing links to relevant country pages are followed and the corresponding country pages are accessed. Thus, data-driven Web exploration (Section 2.2) is used in conjunction with querying structure (navigation along hyperlinks; Section 3). Finally, the actual data is extracted from the country pages, which corresponds to querying contents:[9]

First, the urls of the World Factbook homepage, of the page for countries in Europe, and of a local mirror are defined for the object cia, our "root" object for the Factbook:

```
cia[world →"http://www.odci.gov/cia/publications/nsolo/factbook/global.htm" ;
    europe →"http://www.odci.gov/cia/publications/nsolo/factbook/eur.htm" ;
    mirror →"file:/home/dbis/flogic/data/ciawfb/global.htm" ].
```

To allow for easy substitution of the data source, a generic name cia.src is defined:

```
cia[src →cia.world].      % use the main Factbook page here
```

Alternatively, cia.src may be set to cia.europe, or may even be rule-defined: e.g., if access to cia.world fails, cia.src can be defined as cia.mirror.

The string represented by cia.src is made an instance of class url and accessed via get:

```
cia.src:url.get.          % ⇔ (cia.src):url ∧ (cia.src).get
```

Thus, cia.src.get is the name (logical oid) of the accessed Web document and hrefs@(*label*) is defined for it by the system (unless an error has occurred). The source page cia.src.get of the Factbook contains links to the individual country pages. These links are used to populate the class country with instances C and the urls U of C:

```
C:country[url →U] ←                        % remember the url U of country C after ...
    cia.src.get[hrefs@(Lbl) →→{U}],         % ... extracting all labels Lbl and urls U ...
    pmatch(Lbl, "/(.* ) \([0-9]/","$1", C).   % ... and removing excess parts from Lbl
```

The labels Lbl in cia.src.get are the names of the countries followed by the size of the page in KBytes (e.g., "Spain (32 KB)"). Here, the built-in predicate pmatch is used to strip off this size information: e.g., match("Spain (32 KB)", ..., ..., C) yields C="Spain".

The individual pages of the extracted countries are accessed by defining get for the corresponding urls as usual:

```
U.get ← C:country[url→U].      % retrieve all country pages
```

Observe that the name url can be used for both, the predefined class of urls, and the user-defined method url: In F-logic, also methods and classes are objects; thus, it is possible to reason about schema information.

Finally, data from the country pages can be extracted and stored in the F-logic database. For example, among lots of other data, the following can be extracted (recall that the pmatch predicate treats Web documents as strings):

[9]For a more detailed example in which also data from different sources is integrated, see [HKL+98].

```
C[capital →X]          ←   pmatch(C:country.url.get, "/Capital:.*\n(.*)/", "$1", X).
C[total_area →X]       ←   pmatch(C:country.url.get, "/total area:.*\n(.*sq km)/", "$1", X).
C[external_debt →X]    ←   pmatch(C:country.url.get, "/External debt:.*\n(.*)/", "$1", X).
```

These rules show a strong regularity. Thus, one can take advantage of the meta-programming facilities of F-logic (here: variables at method position) and replace the code by *a single generic rule* and facts describing the used patterns:

```
C[Method→X] ← pattern(Method, RegEx), pmatch(C:country.url.get, RegEx, "$1", X).
pattern(capital,"/Capital:.*\n(.*)/").
pattern(total_area,"/total area:.*\n(.*sq km)/").
pattern(external_debt,"/External debt:.*\n(.*)/").
```

Clearly, such a "pattern-base" may be extended easily for other methods.

Querying the Data. Once the data has been extracted, it can be queried, restructured, and integrated with data from other sources, using all features of F-logic and FLORID, respectively:

```
?– N = count{C ; C:country}.     % (Q1) use aggregation to count the countries
?– C:country[capital→CA].        % (Q2) name all countries and their capitals!
?– C:country, not C.capital.     % (Q3) which countries do not have a capital?
```

For cia.src=cia.world, query (Q1) yields N=266 countries. However, (Q2) outputs a binary relation (Country,Capital) with only 256 entries. (Q3) reveals the 10 "countries" for which the method capital is not defined, e.g., "Antarctica", "Atlantic Ocean", and "World". It turns out that there are some more "countries" which have the method capital defined, yet do *not* have a proper capital. For example, the fact

```
"Bouvet Island":country[capital→"none; administered from Oslo, Norway"]
```

can be derived by FLORID. Thus, we may specify the class of *real* countries as follows:

```
C:real_country ← C:country[capital→CA], not substr("none", CA).
```

Now, the query ?– C:country, not C:real_country discloses 25 more "false countries" (apart from the 10 above) including, e.g., "Bouvet Island", "Clipperton Island", and "Western Sahara".

Schema Browsing and Discovering Structure. Since method and class names are first-class citizens in F-logic, reasoning about schema information is possible. Consider the following queries:

```
?– _:country[M→_].                    % (Q4) what methods are defined for countries?
?– _:country.M, C:country, not C.M.   % (Q5) return countries with undefined methods
```

Query (Q4) yields all single-valued methods (capital, total_area, land_area, etc.) potentially defined for countries (i.e., defined for at least *some* country). The different occurrences of the anonymous (don't care) variable "_" denote *distinct* ∃*-quantified* variables. The first literal _:country.M of (Q5) is a syntactic variant of (Q4); together with the rest of (Q5), "countries" C with undefined methods M are reported: e.g., for C="Ashmore and Cartier Islands", the method M="labor_force" is undefined.

Looking at the CIA World Factbook pages, we discover that important attributes of countries are printed in **boldface**. Hence, we can automatically reveal potentially relevant attributes of countries by extracting all data between ⟨B⟩...⟨/B⟩. Unfortunately, this yields many irrelevant answers. These can be eliminated by *intersecting* the boldface expressions over all country pages. Logically, we use double negation to find the notions which are present on *all* country pages:[10]

```
% for all real countries, extract all bold expressions:
C[bold_ex->>{R}] <- pmatch(C:real_country.url.get, "m!<B>(.*?)</B>!g", "$1", R ).

% what bold expressions are not defined for all real countries?
not_all_bold(M) <- _:real_country[bold_ex->>{M}], C:real_country, not C[bold_ex->>{M}].

% what bold expressions are defined for all real countries?
all_bold(M) <- _:real_country[bold_ex->>{M}], not not_all_bold(M).
```

After evaluating the above program, we obtain the desired answers:

```
?- all_bold(M).
   M/"Location:"
   M/"Description:"
   M/"Area:"
      ⋮
75 output(s) printed
```

4.2 Querying Semantic Tags: A FLORID-XML Parser

With the current state of the art, extracting data from "ordinary" HTML pages requires the often quite tedious and time-consuming task of writing an appropriate wrapper. In particular, this is true when the data source offers only syntactic hints for the *presentation* of the data (e.g., format tags: boldface, italics, etc.) and no information about the *meaning* and/or *context* of data.[11] Thus, a better approach for supporting information gathering from the Web is the use of *semantically meaningful tags*. For example, the *Extensible Markup Language XML* is an effort within W3C to support structured document interchange on the Web [XML97]. XML allows the definition of customized markup languages with application-specific tags. The data model for XML is very simple and corresponds to a tree-like structure; XML documents are (quite verbose) linerizations of this data structure. Since in a *well-formed* XML document

- all tags must be balanced (elements must have both start and end tags present), and
- elements must nest inside each other properly (no overlapping markup),

XML documents have a highly regular structure and can be parsed and analysed very easily.

Consider, for example, the XML representation of a relational database [XML97]: A relational database consists of a set of tables, where each table is a set of records. A record in turn is a set of fields and each field is a pair field-name/field-value. This description of the database suggests a simple nesting of fields inside records inside tables inside databases:

[10] Here, we tacitly assume a stratification (?-sys.strat.doIt.) after each rule.

[11] Nevertheless, as can be seen from the Section 4.1, simple "wrappers" may still be easily specified by FLORID rules (provided the Web source shows enough regularity).

Fig. 5 is an example of a single database in XML with two tables authors and editors.[12] Every element (i.e., expression of the form ⟨*tag* ...⟩...⟨/*tag*⟩) induces a *box*. Based on this box model, a simple XML-parser can be defined very elegantly and concisely in FLORID:

A Simple FLORID-XML Parser. Similar to the access of entry pages of the skeleton extractor (Fig. 3), we first get the root page(s):

```
root[src→→{"http://www.informatik.uni-freiburg.de/~dbis/florid/xml_sample}].
U:url.get ← root[src→→{U}].
```

Once the pages have been accessed, we may query their contents: First, we extract all tags from the documents and populate the class tag with this data:

```
T:tag ← pmatch(_.get,"m!⟨/(\w+)⟩!g","$1",T).
```

Next, we define for every found tag T, the (Perl) regular expression for matching strings of the form ⟨T⟩⟨*data*⟩⟨/T⟩:[13]

```
T[regex→P] ← T:tag, pmatch(T,"/(.*)/","m!⟨$1⟩ (.*?)⟨/$1⟩!gis",P).
```

For every url U whose Web document U.get matches the regular expression T.regex for some tag T, we create a "box" B containing the matched data, and link this box via the method T to the original url U:

```
U[T→→{B:box}] ← T:tag, pmatch(U.get,T.regex, "$1", B).
```

The actual crux of the parser is the *recursive "dissection"* of boxes into sub-boxes and their interlinking by the following rule: For every box B and every tag T, if B contains data matching T.regex, then B is a *complex box* (cbox), and the newly found data NewB is a box, which is accessible from B via T:

```
B:cbox[T→→{NewB:box}] ← B:box, T:tag, pmatch(B,T.regex, "$1", NewB).
```

Finally, provided these rules have been evaluated, we can determine the class of *atomic boxes* (abox), i.e., containing no further sub-boxes:

```
B:abox ← B:box, not B:cbox.
```

The presented simple parser extracts the XML data and maps it into an F-logic database. Using this representation, complex queries can be expressed in a clear and intuitive way with FLORID: For example, for the XML document in Fig. 5 we may be interested in all atomic boxes and the names of their "parent boxes":

```
?- _..X = B:abox.
   _.."address" = "10 Tenth St, Decapolis":abox.
   _.."address" = "2 Second Av, Duo-Duo":abox.
   _.."address" = "1 Premier, Maintown":abox.
   _.."name" = "Robert Roberts":abox.
      ⋮
   _.."born" = "1960/05/26":abox.
   _.."telephone" = "7356":abox.
13 output(s) printed
```

[12]In the actual text document, the author-entries are arranged vertically instead of horizontally.
[13]Here, for simplicity, we do not deal with attributes inside of tags.

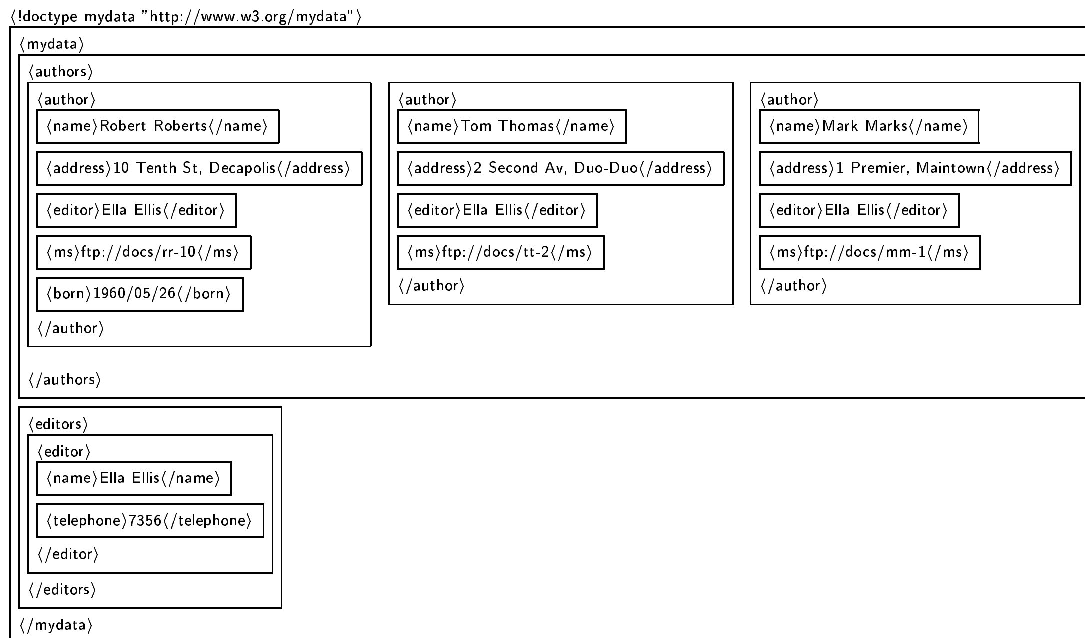

Figure 5: XML representation of a relational database [XML97]

Or we may ask: *"What tags can be found inside of the* authors' *box?"*

```
?– _.."authors"..X.
   _.."authors".."name".
   _.."authors".."address".
   _.."authors".."editor".
   _.."authors".."ms".
   _.."authors".."born".
   _.."authors".."author".
6 output(s) printed
```

Since the above program links direct *and indirect* sub-boxes to the box containing the sub-boxes, also transitive links are present: For example, we can ask for authors' names as follows:

```
?– _.."authors".."name" = X.
   _.."authors".."name" = "Robert Roberts".
   _.."authors".."name" = "Tom Thomas".
   _.."authors".."name" = "Mark Marks".
3 output(s) printed
```

5 Conclusion

We have shown, by means of several illustrative examples, how Web data can be queried in an intuitive and declarative way using FLORID: A generic skeleton extractor has been presented, which allows to automatically extract the hyperlink structure of collections of Web documents. Based on FLORID's logical query language, this structure may be further analysed, e.g., using (general) path expressions and aggregation. In addition to structure-based queries, FLORID also supports content-based queries: In the current implementation,

(Perl) regular expressions are used to work on poorly structured data (e.g., plain text), or to operate on highly structured data (like XML). A general built-in SGML parser is going to be incorporated in the near future and will map SGML documents to F-logic databases.

The extension of FLORID's semantics for querying the Web is described in [HLLS97]; the papers [HKL+98] and [LHL+98] focus on integration of different sources and management of semistructured data with FLORID, respectively. The latter contains also a short introduction to F-logic and path expressions.

References

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 1–18. Springer, 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997.

[BDFS97] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In F. Afrati and P. Kolaitis, editors, *6th Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 336–350, Delphi, Greece, 1997. Springer.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrandt, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1996.

[BRR97] F. Bry, K. Ramamohanarao, and R. Ramakrishnan, editors. *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, Montreux, Switzerland, 1997. Springer.

[Bun97] P. Buneman. Semistructured Data (invited tutorial). In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 117–121, Tucson, Arizona, 1997.

[DBL] DBLP Computer Science Bibliography. `http://www.informatik.uni-trier.de/~ley/db/`.

[FLO] The FLORID Home Page. `http://www.informatik.uni-freiburg.de/~dbis/florid/`.

[GMNP97] F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Datalog++: A Basis for Active Object-Oriented Databases. In Bry et al. [BRR97].

[HKL+98] R. Himmeröder, P.-T. Kandzia, B. Ludäscher, W. May, and G. Lausen. Search, Analysis, and Integration of Web Documents: A Case Study with FLORID. In *Proc. Intl. Workshop on Deductive Databases and Logic Programming (DDLP'98)*, pp. 47–57, Manchester, UK, 1998. GMD Report 22.

[HLLS97] R. Himmeröder, G. Lausen, B. Ludäscher, and C. Schlepphorst. On a Declarative Semantics for Web Queries. In Bry et al. [BRR97], pp. 386–398.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with *FLORID*: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8), 1998. to appear.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *Proc. Sixth International Workshop on Research Issues in Data Engineering (RIDE)*, 1996.

[MV98] U. Masermann and G. Vossen. Suchmaschinen und Anfragen im World Wide Web. *Informatik Spektrum*, 21(1):9–15, 1998.

[Suc97] D. Suciu, editor. *Proc. of the Workshop on Management of Semi-Structured Data (in conjunction with SIGMOD/PODS)*, Tucson, Arizona, 1997. `http://www.research.att.com/~suciu/workshop-papers.html`.

[XML97] Extensible Markup Language (XML). `http://www.w3.org/XML/`, 1997.