Pergamon

Information Systems Vol. 23, No. 8, pp. 1–25, 1998

Printed in Great Britain
0306-4379/98 $19.00 + 0.00

MANAGING SEMISTRUCTURED DATA WITH FLORID: A DEDUCTIVE OBJECT-ORIENTED PERSPECTIVE

Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May and Christian Schlepphorst

Institut für Informatik, Universitätsgelände Flugplatz, D-79085 Freiburg, Germany
{ludaesch,himmeroe,lausen,may,schlepph}@informatik.uni-freiburg.de

(Received 17 February 1998; in final revised form 19 October 1998)

Abstract — The closely related research areas *management of semistructured data* and languages for *querying the Web* have recently attracted a lot of interest. We argue that languages supporting deduction and object-orientation (DOOD languages) 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.
We elaborate on the management of semistructured data and show how reachability queries involving general path expressions and the extraction of data paths in the presence of cyclic data can be handled. We then propose a formal model for querying structure and contents of Web data and present its declarative semantics. A main advantage of our approach is that it brings together the above-mentioned issues in a unified, formal framework and—using the FLORID system—supports rapid prototyping and experimenting with all these features. Concrete examples illustrate the concise and elegant programming style supported by FLORID and substantiate the above-mentioned claims.

1. INTRODUCTION

Models and languages for *semistructured data* (*ssd*) have recently attracted a lot of interest [Abi97, AQM+97, BDHS96, Suc97]. One motivation for studying *ssd* is the immense growth and impact of the *World-Wide Web*, a new "drosophila" for studying issues related to semistructured data. Moreover, there is a growing need for integration of data from heterogeneous sources (e.g., legacy systems or data available from the Web), for which *ssd* provides a common data model. 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].

The Web is the most immediate and probably also the most exciting example for *ssd*: almost all degrees of structure from highly structured data to completely chaotic data happily coexist in the Web—often only a mouse click apart. The main techniques for accessing Web data are *browsing*, i.e., "manual" navigational access, and *searching*, where the potentially relevant pages are found using search engines (which act as keyword indexes, e.g., AltaVista or HotBot) or Web guides (like InfoSeek, Excite, or Yahoo) with their predefined categories. In both cases, Web exploration is data-driven, i.e., the continuation of a search depends on the information which has been acquired so far. Clearly, *navigation* is an important issue for querying semistructured data in general, and querying the Web in particular.

In order to overcome the limitations of simple browsing and searching, several *Web query languages* and systems have been proposed which allow to extract and reorganize data from the Web. Since the Web offers access to an enormous number of database servers, it also opens a challenging application field for approaches to *information integration*, where the task is to provide uniform access to heterogeneous information sources. Most of the common integration architectures comprise *wrappers* for translating from different local languages into a common language shared between all sources, and *mediators* which provide—on a higher conceptual level—an integrated

view on the heterogeneous data. Since rules are a powerful means for defining views, several approaches for mediator specification make use of rule-based languages. On top of the mediator language, there may be a further *user language* for querying the integrated data.

Since the field is rapidly growing and changing, semantically rich, flexible, and adaptable frameworks are required to cope with the evolving needs. In this paper, we argue that languages supporting deduction and object-orientation (DOOD languages) are particularly suited in this context: *Object-orientation* provides a flexible and rich data model, e.g., 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 are just (very simple) instances of the former. *Deductive rules* provide a powerful framework for expressing complex queries in a high-level, declarative programming style, e.g., for specifying mediators, or for data-driven Web exploration.

Thus, a main advantage of the DOOD approach is that it integrates all above-mentioned issues related to semistructured data in a unified, formal framework. We substantiate this claim using FLORID, a prototype implementing the declarative DOOD language F-logic. The sytem can be used for rapid prototyping of applications, e.g., for extracting, analyzing, and restructuring Web data, possibly attuning it with information from other sources. Since FLORID also contains built-ins for extracting data from Web documents, an impedance mismatch due to separate languages for wrappers, mediators, and user queries is avoided. Clearly, specialized languages can be much more efficient in solving their specific tasks; however, the strength of our approach lies in its easy extensibility by means of rules. For example, in a first stage, a certain method for analyzing data or navigation on semistructured data may be defined by rules without changing the system. Then, when this method is recognized as central to an application it may be provided as built-in by the system.

Outline

The paper is structured as follows: The basic formal framework of F-logic with path expressions is introduced in Section 2. In Section 3, we elaborate on the representation and management of semistructured data using F-logic. For navigation, a powerful and intuitive language for general path expressions and its executable specification is presented. In Section 4, we propose a formal model for querying structure and contents of Web data and present its declarative semantics. An example which illustrates how Web data is queried and restructured with FLORID is given in Section 5. Section 6 contains a short conclusion and outlook.

Related Work

In order to overcome the limitations of simple browsing and searching, several *Web query languages* and systems have been proposed which allow to extract and reorganize data from the Web: e.g., the SQL extensions WebSQL [MMM96] and W3QL [KS95], the languages from the ARANEUS project [ARA98], or PIQL [LSCS97], a language incorporating ideas from information retrieval. More closely related to our framework are WebLog [LSS96] and ADOOD [GMNP97], logic-based languages borrowing from ideas of F-logic. However, [LSS96] and [GMNP97] do not provide a formal semantics for Web access and navigation on semistructured data, as we do. Abiteboul and Vianu [AV97] introduce theoretical foundations to investigate the computability of Web queries, based on so-called *Web machines*, but do not present a concrete Web query language. Mendelzon and Milo [MM97] present a formal model of Web queries which is closely related to [AV97], in which especially the effects of limited access to data and the lack of concurrency are discussed.

In [Bun97] and, more detailed in [Abi97], basic issues and notions related to semistructured data are introduced. Two prominent query languages for semistructured data are Lorel [AQM+97] and UnQL [BDHS96]. Both use a data model based on labeled graphs. Lorel is an extension of OQL [Cat94], providing additional features such as coercion and general path expressions. The use of coercion relieves the user from specifying the precise types of objects. An important feature of UnQL is a construct called *traverse* that allows tree restructuring up to arbitrary depth. UnQL can be translated into UnCal, a calculus enabling certain optimization techniques. Path

expressions as used in languages for semistructured data (see, e.g., [CCM96, AQM+97, FS98]) have been considered earlier in the context of object-oriented databases (e.g., in [KKS92, VdBV93, GPVdBVG94, FLU94]).

Semistructured data also arises in the context of information integration. In the TSIMMIS project [TSI98], the Mediator Specification Language (MSL) [PAGM96, PGMU96] is proposed, a variant of Datalog extended with "semantic oid's". MSL uses information provided by wrappers, which represent their data in the Object Exchange Model (OEM). Due to its object-oriented features, F-logic may also be used for information integration and mediation: some approaches use F-logic directly, e.g., [CRD94, LBT92], whereas others use specialized variants [LSS93].

2. THE DOOD FRAMEWORK OF FLORID

FLORID[1] is a prototypical deductive object-oriented database system, which has been recently extended to provide declarative rule-based access for querying the Web [HLLS97, HLLS98]. FLORID's formal foundation is *F-logic* [KLW95], a prominent DOOD language providing complex objects, rule-defined classes and signatures, multiple inheritance, and a uniform handling of data and metadata. In particular, reasoning about schema information is supported using variables at method and class positions. In this section, we briefly review the basic features of F-logic and its extension by path expressions [FLU94] as needed for our exposition.

2.1. Syntax and Object Model of F-logic

- *Symbols*: The F-logic alphabet comprises sets $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{V}$ of *object constructors* (i.e., function symbols), predicate symbols (including $\doteq$), and variables, respectively. Variables are denoted by capitalized symbols (e.g., $X, Name$), whereas all other symbols, especially constants (0-ary object constructors) are denoted in lowercase (e.g., $a, john$). An expression is called *ground* if it involves no variables. In addition to the usual first-order connectives and symbols, there are a number of special symbols[2]:], [, }, {, $\rightarrow$, $\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!>$, :, ::.

- *Id-Terms/Oids*:

  (0) First-order terms over $\mathcal{F}$ and $\mathcal{V}$ are called *id-terms*, and are used to name objects, methods, and classes. Ground id-terms correspond to *logical object identifiers* (*oid*s), also called object *names*.

- *Atoms*: Let O, M, R_i, X_i, C, D, T be id-terms. In addition to the usual first-order atoms like $p(X_1, \ldots, X_n)$, there are the following basic types of atoms:

  (1) $O[M \rightarrow R_0]$ (2) $O[M \twoheadrightarrow \{R_1, \ldots, R_n\}]$ (3) $C[M \Rightarrow T]$ (4) $C[M \Rightarrow\!\!> T]$.

  (1) and (2) are *data atoms*, specifying that a *method* M applied to an object O yields the result object R_i. In (1), M is a *single-valued* (or *scalar*) method, i.e., there is at most one R_0 such that $O[M \rightarrow R_0]$ holds. In contrast, in (2), M is *multi-valued*, so there may be several result objects R_i. For $n = 1$ the braces may be omitted.
  (3) and (4) denote *signature atoms*, specifying that the (single-valued and multi-valued, respectively) method M applied to objects of *class* C yields results of type T.

  The organization of objects in classes is specified by *isa-atoms*:

  (5) $O : C$ (6) $C :: D$.

  (5) defines that O is an *instance* of class C, while (6) specifies that C is a *subclass* of D.

- *Parameters*: Methods may be *parameterized*, so $M@(P_1, \ldots, P_k)$ is allowed in (1)–(4), where $P_1, \ldots, P_k$ are id-terms; e.g., john[salary@(1998)→50000].

[1] *F-LO*gic *R*easoning *I*n *D*atabases; available from [FLO98].
[2] We do not deal with inheritance in this paper, so we omit the symbols for *inheritable* methods [KLW95].

- *Programs*: F-logic *literals*, *rules*, and *programs* are defined as usual, based on F-logic atoms.

As a concise notation for several atoms specifying properties of the same object, *F-molecules* are used: e.g., instead of john:person∧john[age→31]∧john[children↠{bob,mary}], we may simply write john : person[age→31; children↠{bob,mary}].

2.1.1. Object Model

An *F-logic database (instance)* is represented by a set of facts (i.e., atoms and molecules). Object-oriented databases often admit a natural graph-like representation, which is, e.g., exploited in GOOD [GPVdBVG94]. Similarly, F-logic databases can be represented as labeled graphs where nodes correspond to logical oids, and where the different kinds of labeled edges (→, ↠, ⇒, ⇒⇒) are used to represent different relations among objects (at the class or instance level).

Example 1 (Publications Database) In Figure 1, a fragment of an object-oriented publications database is depicted. The uppermost part shows classes (rectangles) linked via double-shafted arrows (denoting the signatures of methods). Below, individual objects (oval nodes) are depicted. Instance and subclass relationships are denoted by dotted and dashed edges, respectively. At the instance level, methods are denoted by edges which are labeled with the respective method names; following F-logic's notation, edges are single-headed and double-headed for single-valued and multi-valued methods, respectively. The F-logic equivalent of the graph is sketched in the lower part of the figure.

2.2. Path Expressions in F-logic

In addition to the basic F-logic syntax, the Florid system also supports *path expressions* to simplify object navigation along single-valued and multi-valued method applications and to avoid explicit join conditions [FLU94]. The basic idea is to allow the following *path expressions* wherever id-terms are allowed:[3]

$$(7)\ O.M \qquad\qquad (8)\ O..M$$

The path expression in (7) is *single-valued* and refers to the unique object R_0 for which $O[M{\rightarrow}R_0]$ holds; (8) is a *multi-valued* path expression and refers to each R_i for which $O[M{\twoheadrightarrow}\{R_i\}]$ holds. O and M may be id-terms or path expressions; moreover, M may be parameterized, i.e., of the form $M@(P_1,\ldots,P_k)$.

In order to obtain a unique syntax and to specify different orders of method applications, parentheses are used: By default, path expressions associate to the left, so $a.b.c$ is equivalent to $(a.b).c$ and specifies the unique object o such that $a[b{\rightarrow}x] \wedge x[c{\rightarrow}o]$ holds (note that $x = a.b$). In contrast, $a.(b.c)$ is the object o' such that $b[c{\rightarrow}x'] \wedge a[x'{\rightarrow}o']$ holds (here, $x' = b.c$); generally, $o' \neq o$. Note that in $(a.b).c$, b is a method name, whereas in $a.(b.c)$ it is used as an object name. Further note that function symbols can also be applied to path expressions, since path expressions (like id-terms) are used to reference objects.

As path expressions and F-logic atoms may be arbitrarily nested, a concise and very flexible specification language for object properties is obtained:

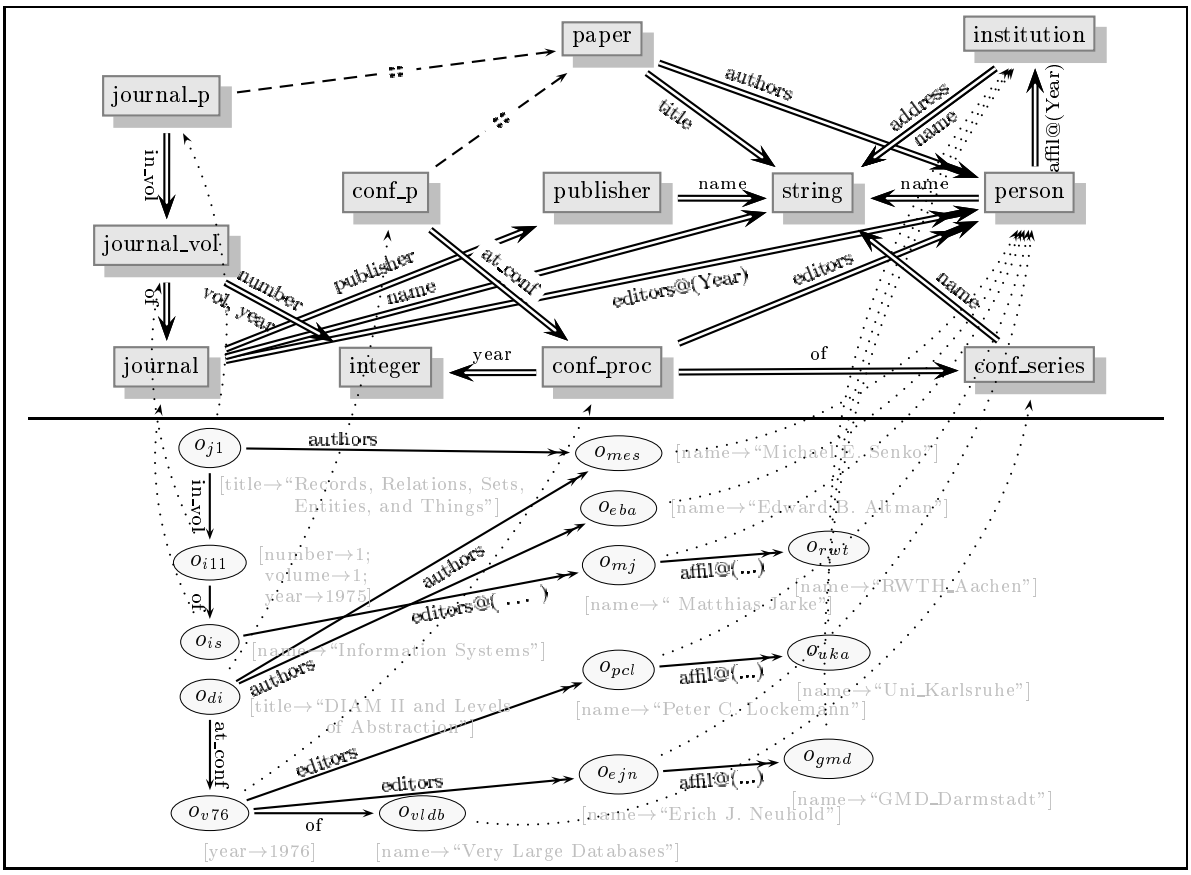
Example 2 (Path Expressions) Consider again the schema given in Figure 1. Given the name n of a person, the following path expression references all editors of conferences in which n had a paper:[4]

_ : conf_p[authors↠{_[name→n]}].at_conf..editors

Therefore, the answer to the *query*

[3] However, to avoid semantical problems, the use of path expressions in rule heads is restricted; see Section 2.2.2.

[4] Each occurrence of "_" denotes a distinct don't-care variable (which is existentially quantified at the innermost level).



paper[authors⇒↠person; title⇒string]. conf_p :: paper. journal_p :: paper.
journal_p[in_vol⇒volume]. conf_p[at_conf⇒conf_proc].
journal_vol[of ⇒journal; volume⇒integer; number⇒integer; year⇒integer].
journal[name⇒string; publisher⇒string; editors@(integer)⇒↠person].
conf_proc[of_conf⇒conf_series; year⇒integer; editors@(integer)⇒↠person].
conf_series[name⇒string]. publisher[name⇒string].
person[name⇒string; affil@(integer)⇒institution]. institution[name⇒string; address⇒string].

---

o_{j1} : journal_p[title→"Records, Relations, Sets, Entities, and Things"; authors↠{o_{mes}}; in_vol→o_{i11}].
o_{di} : conf_p[title→"DIAM II and Levels of Abstraction"; authors↠{o_{mes}, o_{eba}}; at_conf→o_{v76}].
o_{i11} : journal_vol[of→o_{is}; number→1; volume→1; year→1975].
o_{is} : journal[name→"Information Systems"; editors@(...)↠{o_{mj}}].
o_{v76} : conf_proc[of→vldb; year→1976; editors↠{o_{pcl}, o_{ejn}}].
o_{vldb} : conf_series[name→"Very Large Databases"].
o_{mes} : person[name→"Michael E. Senko"]. o_{mj} : person[name→"Matthias Jarke"; affil@(...)→o_{rwt}].
o_{rwt} : institution[name→"RWTH_Aachen"]. ...

Fig. 1: Publications Object Base: Schema/Instance Data in Graph and F-logic Representation

?- P : conf_p[authors↠{_[name→n]}].at_conf[editors↠{E}].

is the set of all pairs (P,E) such that P is (the logical oid of) a paper written by n, and E is the corresponding proceedings editor. If one is interested in the affiliations of the above editors at publishing time, the query can be adjusted easily:

?- P : conf_p[authors↠{_[name→n]}].at_conf[year→Y]..editors[affil@(Y)→A].

Thus, FLORID's path expressions support navigation and specification of object properties along two dimensions: the "depth" dimension corresponds to navigation along method applications (".")

and ".."), while the bracketed specification list of molecules defines properties of intermediate objects in the "breadth" dimension. Note that constraints *within* the expressions can be stated using variables.

In order to access objects which are implicitly defined in the middle of path expressions, one may define the method self by $x[\mathsf{self}{\rightarrow}x]$ for all relevant objects x. Then one can simply write $\ldots[\mathsf{self}{\rightarrow}O]\ldots$ anywhere in a complex path expression in order to bind the name of the current object to the variable O.[5]

Example 3 (Path Expressions with self) Recall the second query in Example 2. If the user is also interested in the respective conferences, the query can be reformulated as

?- P : conf_p[authors↠{_[name→n]}].at_conf[self→C; year→Y]..editors[affil@(Y)→A].

2.2.1. References: Truth Value vs. Object Value

Id-terms, F-logic atoms, and path expressions can all be used to reference objects. This is obvious for id-terms (0) and path expressions (7–8). Similarly, F-logic atoms (1–6) not only have a truth value, but also reference objects, i.e., yield an object value. For example, $o : c[m{\rightarrow}r]$ is a reference to o, additionally specifying o's membership in c and its value for m.

Consequently, all F-logic expressions of the form (0–8) are called *references*. F-logic references have a dual reading: Given an F-logic database $\mathcal{I}$ (see below), a reference has

- an *object value*, which yields the name(s) of the objects reachable in $\mathcal{I}$ by the corresponding reference, and
- a *truth value* like any other literal or molecule of the language; in particular, a reference r evaluates to *false* if there exists no object which is referenced by r in $\mathcal{I}$.

Thus, a path expression may be conceived as a logical formula (*deductive perspective*), or as a name for a number of objects (*object-oriented perspective*).

Consider the following path expression and its equivalent (wrt. the truth value) flattened rewriting:

$$a..b[c{\twoheadrightarrow}\{d.e\}] \quad \Leftrightarrow \quad a[b{\twoheadrightarrow}\{X_{ab}\}] \wedge d[e{\rightarrow}X_{de}] \wedge X_{ab}[c{\twoheadrightarrow}\{X_{de}\}]. \qquad (*)$$

In this way, using the flattened equivalent, the truth value of arbitrarily complex path expressions in the *body* of rules can be easily determined. The object values *obj* of a path expression are the names of the referenced objects: e.g., for (∗) we have

$$obj(a..b) = \{x_{ab} \mid \mathcal{I} \models a[b{\twoheadrightarrow}\{x_{ab}\}]\} \qquad \text{and} \qquad obj(d.e) = \{x_{de} \mid \mathcal{I} \models d[e{\rightarrow}x_{de}]\}\ ,$$

where $\mathcal{I} \models \varphi$ means that φ holds in $\mathcal{I}$. Observe that $obj(d.e)$ contains at most one element because the *single-valued* method e is applied to the unique oid d. In general, for an F-logic database $\mathcal{I}$, the object values of ground expressions are given by the following mapping *obj* from ground references to sets of ground references:

$$\begin{array}{rcll}
obj(t) & := & \{t' \mid t' = t \text{ and } \mathcal{I} \models t'\}, \text{ for a ground id-term } t & (0)\\
obj(o[\ldots]) & := & \{o' \in obj(o) \mid \mathcal{I} \models o'[\ldots]\} & (1), \ldots, (4)\\
obj(o : c) & := & \{o' \in obj(o) \mid \mathcal{I} \models o' : c\} & (5)\\
obj(c :: d) & := & \{c' \in obj(c) \mid \mathcal{I} \models c' :: d\} & (6)\\
obj(o.m) & := & \{r' \in obj(r) \mid \mathcal{I} \models o[m{\rightarrow}r]\} & (7)\\
obj(o..m) & := & \{r' \in obj(r) \mid \mathcal{I} \models o[m{\twoheadrightarrow}\{r\}]\} & (8)
\end{array}$$

Observe that $obj(t) = \emptyset$ for a ground id-term t which does not hold (i.e., occur) in $\mathcal{I}$. Conversely, a ground reference r is called *active* if $obj(r) \neq \emptyset$. r can be classified as either single-valued or multi-valued:

[5] A similar feature is used in many other languages, e.g., in XSQL [KKS92] and in Lorel [AQM+97].

- r is called *multi-valued* if
  - it has the form $o..m$, or
  - it has one of the forms $\underline{o}[\ldots]$, $\underline{o}:c$, $\underline{c}::d$, or $\underline{o.m}$, and any of the underlined subexpressions is multi-valued;
- in all other cases, r is *single-valued*.

2.2.2. Object Creation

Single-valued references can create *anonymous objects* when used in the *head* of rules (this feature will be crucial for our approach to data-driven Web exploration with FLORID):

Example 4 (Spouse Invention)
Consider the following rule with a single-valued path expression in the head:

$$\mathsf{X.spouse:person} \leftarrow \mathsf{married(X:person)}.$$

For every married person X, a new object X.spouse is created and inserted into class person (if X[spouse→Y] already holds, then X.spouse=Y). Note the seamless and natural integration of Datalog atoms (married(...)) with F-logic constructs (X:person).

In contrast to single-valued references, multi-valued references may lead to semantical problems and ambiguities when used in the head:[6] e.g., the meaning of the following facts is unclear:

$$\mathsf{john..children[age{\rightarrow}24]}. \qquad \mathsf{john..children[lives\_in{\rightarrow}berlin]}.$$

It is unclear how many children of John are 24 years old: at least one, or all? Do all of them live in Berlin, or only those who are 24 years old? To avoid such problems, we disallow the use of multi-valued references in the head of rules.

2.3. F-logic Semantics

For simplicity, we confine ourselves to the features of F-logic with path expressions which are essential for our exposition; see [KLW95] for details on inheritance, typing, and negation. Similar to first-order logic, formulas are interpreted over semantic structures in F-logic. As usual, we focus on special *Herbrand-style structures*, called *H-structures*[7] over a suitable universe:

As usual, given the set $\mathcal{F}$ of object constructors, the Herbrand universe U ($=U(\mathcal{F})$) consists of all ground id-terms over $\mathcal{F}$. The Herbrand base $\mathcal{HB}$ consists of all ground atoms over U, and every subset of $\mathcal{HB}$ induces a Herbrand interpretation.

In the presence of path expressions, U is not sufficient as H-universe since single-valued path expressions can create new logical oids when used in rule heads. Thus, single-valued path expressions behave like Skolem function symbols, and U has to be closed wrt. application of single-valued path constructors; the resulting closure is denoted by $U^\star$:

Definition 1 (H-Universe, H-Base)
Let $\mathcal{F}$ and $\mathcal{P}$ denote the sets of object constructors and predicate symbols, respectively.

- The *(extended) H-universe* $U^\star$ ($=U^\star(\mathcal{F})$) is the least set $U^\star \supseteq U$ such that:
  - if $m, t_0, \ldots, t_n \in U^\star$, then $(t_0.m)$ and $(t_0.m@(t_1, \ldots, t_n))$ are in $U^\star$ for every $n \in \mathsf{IN}$.

  The elements of $U^\star$ are called *pure references*.
- The *(extended) H-base* $\mathcal{HB}^\star$ ($=\mathcal{HB}^\star(\mathcal{F},\mathcal{P})$) consists of all ground atoms (i.e., Datalog, F-logic, and $\doteq$ atoms) and ground pure references over $\mathcal{P}$ and $U^\star$.

[6] See the section *scalarity and well-formedness* of references in [FLU94] for details.

[7] H-structures can be mapped to *F-structures*, whose domain are the equivalence classes of equality on the H-universe [KLW95, Sec. 9].

Note that $U^\star$ and $\mathcal{HB}^\star$ are generally infinite, i.e., provided the language comprises at least one constant t_0.

Since $\mathcal{HB}^\star$ contains the set $U^\star$ of pure references, every subset of $\mathcal{HB}^\star$ also assigns a truth value to every $r \in U^\star$. Due to the inherent semantics of the class hierarchy and object identity, we are only interested in interpretations which satisfy certain properties:

Definition 2 (H-Interpretation) A subset $\mathcal{H} \subseteq \mathcal{HB}^\star$ is an *H-interpretation* if it satisfies the following axioms for all pure references $c, d, e, o, m, r, r' \in U^\star$:

- The *closure axioms* of F-logic (cf. [KLW95, Sec. 7]), restricted to active names:
  - $c :: c \in \mathcal{H}$, (*subclass reflexivity*)
  - if $c :: d \in \mathcal{H}$ and $d :: e \in \mathcal{H}$ then $c :: e \in \mathcal{H}$, (*subclass transitivity*)
  - if $c :: d \in \mathcal{H}$ and $d :: c \in \mathcal{H}$ then $c \doteq d \in \mathcal{H}$, (*subclass acyclicity*)
  - if $o : c \in \mathcal{H}$ and $c :: d \in \mathcal{H}$ then $o : d \in \mathcal{H}$, (*subclass inclusion*)
  - if $o[m \rightarrow r] \in \mathcal{H}$ and $o[m \rightarrow r'] \in \mathcal{H}$ then $r \doteq r' \in \mathcal{H}$, (*scalarity of "→"*)
  - reflexivity, transitivity, and symmetry of $\doteq$, substitutability of equal objects. (*equality axioms*)
- The *path expression axioms*:
  - if $o.m \in \mathcal{H}$ then $o[m \rightarrow o.m] \in \mathcal{H}$,
  - if $o[m \rightarrow r] \in \mathcal{H}$ then $o.m \doteq r \in \mathcal{H}$.
- The *active name axiom*:
  - for every $r \in \mathcal{H}$ and every $r' \in U^\star$ occurring in r: $r' \in \mathcal{H}$.

(analogously for $m@(t_1, \ldots, t_n)$ instead of m).

The last axiom allows a simple declaration of the set $U_\mathcal{H}$ of *active names* of $\mathcal{H}$, namely $U_\mathcal{H} := U^\star \cap \mathcal{H}$. For an arbitrary $\mathcal{I} \subseteq \mathcal{HB}^\star$, we denote by $\mathcal{C}\ell(\mathcal{I})$ the closure of $\mathcal{I}$ wrt. the above axioms.

Example 5 The active names of the closure of $\mathcal{I} := \{\mathsf{john[wife{\rightarrow}jane]}, \mathsf{jane[husband{\rightarrow}john]}\}$ are:
$U^\star \cap \mathcal{C}\ell(\mathcal{I}) = \{\mathsf{john(.wife.husband)}^n\mathsf{(.wife)}^{\{0,1\}}, \mathsf{jane(.husband.wife)}^n\mathsf{(.husband)}^{\{0,1\}} \mid n \in \mathsf{I\!N}\}$.
Moreover, $\mathcal{C}\ell(\mathcal{I})$ includes, e.g., the sets
$\{\mathsf{jane(.husband.wife)}^n\mathsf{[husband{\rightarrow}jane(.husband.wife)}^n\mathsf{.husband}] \mid n \in \mathsf{I\!N}\}$ and
$\{\mathsf{jane(.husband.wife)}^n\mathsf{.husband} \doteq \mathsf{john} \mid n \in \mathsf{I\!N}\}$, and many equality atoms.

In the Florid system, Herbrand structures are represented as F-structures ([KLW95, Sec. 9]), i.e., active names are mapped to *equivalence classes* of objects. In this way, a finite database $\mathcal{I}$ whose closure results in an infinite H-interpretation $\mathcal{C}\ell(\mathcal{I})$ can still be represented by a finite F-structure. For instance, for $\mathcal{I}$ from Example 5, the equivalence class for john subsumes all active names $\{\mathsf{john(.wife.husband)}^n \mid n \in \mathsf{I\!N}\}$, and can be represented by the equalities $\mathsf{john.wife} = \mathsf{jane}$ and $\mathsf{jane.husband} = \mathsf{john}$.

The semantics of a (negation-free) F-logic program P can now be defined as the minimal H-interpretation $\mathcal{H}$ for which $\mathcal{H} \models P$, called the *canonical model* of P.[8] The standard semantics of Florid is *inflationary* [AHV95] and thus always yields a unique **PTIME**-computable model provided no new oids are invented (or at most a fixed number of them, independent of the given database). Additionally, Florid supports user-stratified negation which allows a more natural definition of certain queries.[9] In [MLL97], it has been shown how F-logic programs with negation can be evaluated in Florid wrt. well-founded semantics via a program rewriting technique.

[8] The problem of choosing a canonical model in the presence of negation is dealt with in [KLW95].

[9] E.g., the complement of transitive closure is trivial to specify with stratified semantics but not so with inflationary semantics.

3. MANAGING SEMISTRUCTURED DATA

Object-oriented data models like the one of F-logic provide flexible modeling capabilities due to their rich structure, e.g., class hierarchies, (possibly parameterized) single-valued and multi-valued methods, and strict typing constraints. However, data is often available only in unstructured or semistructured form, especially, when retrieved from the Web. Not surprisingly, the enormous success of the Web has recently lead to an increasing interest in models and languages for *semistructured data* (*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 this section, we recast a prototypical model for *ssd* in F-logic and illustrate how *general path expressions* (used for reachability queries on *ssd*) may be specified and implemented in an elegant and declarative way within a DOOD framework. Moreover, we show how derived equalities may be used in FLORID for extracting data paths on cyclic structures.

3.1. Semistructured Data Model

Since in semistructured data there may be very little structure, or the structure is contained in the data and has to be discovered, 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 3 (Semistructured Database) Let $\mathcal{N}$ be a set of *nodes* and $\mathcal{L}$ a set of *labels*. A *semistructured database*[1] (*ssdb*) $\mathcal{D}$ is a finite subset of the set of *labeled edges* $\mathcal{E} = \mathcal{N} \times \mathcal{L} \times \mathcal{N}$.

The main differences between different models for *ssd* concern the kind of data stored in nodes and labels, and the location where the data is stored: One can classify data of a *ssdb* very roughly as *simple data* (e.g., `1970`, `"E. F. Codd"`) or *schema-like* data, i.e., corresponding to attribute or class names (e.g., `year`, `person`). In the terminology of [Bun97], the latter are called *symbols*. Note however that the distinction between simple and schema data is often blurred. Therefore, we adopt a very generic, initially untyped framework, where labels are strings and where nodes may or may not hold additional data. In the former case, nodes are called *complex*, otherwise *opaque*.

The other difference concerns the location where the data is stored. For example, in the *ssd* model of UnQL [BDHS96], all information (simple data and symbols) is contained in labels only. In contrast, in the OEM data model underlying Lorel, labels contain symbols, whereas simple data is stored in leaves (internal nodes correspond to oids without further information) [AQM+97, Bun97]. In contrast, we do not impose any restrictions on the location of data, i.e., nodes may be complex or opaque.

Semistructured Data in F-logic

The graph representation of F-logic databases (cf. Figure 1) contains the *ssd* model as a *special case*: 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:

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

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

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

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.

A proverbial example for *ssd* is the link structure of a set of Web documents: nodes correspond to Web documents and edges to labeled hyperlinks between documents. In this case, if nodes are

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

opaque, i.e., when no information apart from the link structure is available, we speak of the *(Web) skeleton*[2] of a set of Web documents:

Example 6 (DBLP Skeleton) Consider the labeled graph depicted in Figure 2 which represents a fragment of the skeleton of the DBLP server [DBL98]: 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*. Figure 8 depicts a generic skeleton extractor for Florid.

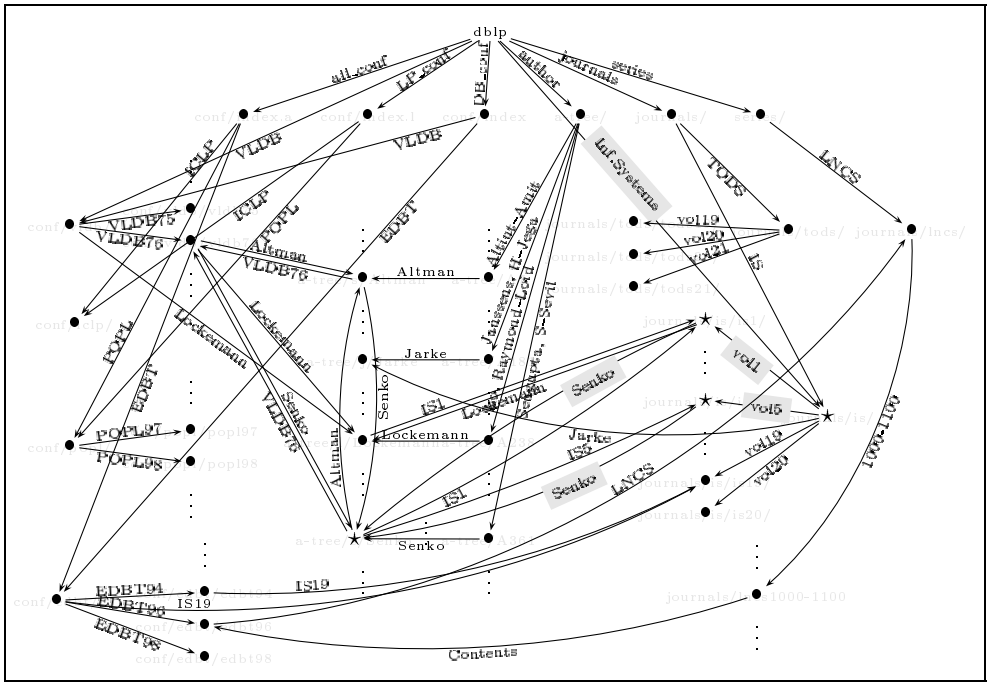


Fig. 2: Fragment of the Skeleton of the DBLP Database

3.2. Querying Semistructured Data

Relational databases can be queried using formalisms like relational algebra or calculus, or deductive rules in the style of Datalog. In contrast, semistructured databases lend themselves to a navigational access where one starts at a distinguished root node x_o and tries to find all nodes reachable from x_o such that the encountered labels match a given sequence of labels. Since *ssdb*'s may be considered as specialized object-oriented databases, languages for navigating in object-oriented databases are also applicable in the context of *ssd*. In particular, F-logic path expressions as described in Section 2.2 can be employed without change:

3.2.1. Simple Path Expressions

Following the terminology of Lorel [AQM+97], an object name x_0 followed by a sequence of labels $\ell_1, \ldots, \ell_n$ is called a *simple path expression*. Moreover, a *data path* is a sequence $(x_0, \ell_1, x_1, \ell_2, \ldots, x_n)$ such that $x_{i-1} \xrightarrow{\ell_i} x_i$ is a labeled edge in the given *ssdb* for all $i = 1, \ldots, n$. The F-logic equivalent of a simple path expression is given by the following multi-valued reference:

$$x_0..\ell_1..\ell_2.. \cdots ..\ell_n \qquad (**)$$

[2] For a completely different "Web skeleton" see `http://www.m-w.com/mw/art/skeleton.htm`.

The dual (truth-value and object-value) semantics of (**) is given by the semantics for path expressions in F-logic (Section 2.2). Clearly, there are in general several data paths which match a given (simple) path expression. In contrast, if a *single-valued* path expression of the form $x_0.\ell_1.\ell_2.\ \cdots\ .\ell_n$ were used, at most one data path would be defined.

Using the method self (defined by $X[\mathsf{self}{\rightarrow}X]$, cf. Example 3), the intermediate objects $X_1, \ldots, X_n$ in the data path can be returned in the query answer:[3]

$$x_0..\ell_1[\mathsf{self}{\rightarrow}X_1]..\ell_2[\mathsf{self}{\rightarrow}X_2]..\ \cdots\ ..\ell_n[\mathsf{self}{\rightarrow}X_n]. \qquad (***)$$

Note that the above simple path expression (**) is valid F-logic syntax (Section 2.2) even when used with *variables* in place of x_0 and $\ell_1, \ldots, \ell_n$:

Example 7 (Querying the DBLP Skeleton) Given the DBLP skeleton (see Figure 2), we may ask for all labels ℓ reachable from the root node dblp after a link labeled "Inf. Systems" such that the corresponding page contains a link to a given person P, say "Michael E. Senko" (the author of the very first paper in *Information Systems*):

```
?- dblp.."Inf. Systems"..L.."Michael E. Senko".
```

For the DBLP skeleton, FLORID returns two answers L="Volume 1, 1975" and L="Volume 5, 1980". The object value of the path expression is the url of the author's page at DBLP.

The main power of such path expressions becomes apparent when more information than merely the Web skeleton is available, i.e., when nodes are not opaque but contain additional information about the object. Such additional information may be extracted from Web documents by querying their *contents* (Section 4.1), or may stem from another information source or the local database. For example, the FLORID query

```
?- dblp.."Inf. Systems"..L..P,  substr("Volume",L),
P:person.spouse[lives_in→P.lives_in].
```

returns all persons P who (i) have a paper in a volume of IS, and (ii) live in the same place as their spouse.

As illustrated by this example, (simple) path expressions may be combined freely with other F-logic constructs (single-valued path expressions, parameterized methods, class information, etc.), thereby obtaining a very powerful language. The seamless integration of path expressions into the DOOD framework of F-logic is possible mainly due to their dual reading, i.e., as truth-valued expressions (*deductive perspective*) and as object-valued expressions (*object-oriented perspective*).

Computing Cycle-Free Data Paths

Path expressions are used to express reachability queries, i.e., for extracting the set of nodes which are reachable from a source node via a sequence of labels. Therefore, on finite structures, the answer to a reachability query is finite, even on *ssdb*'s with cycles. On the other hand, for finite structures there may be infinitely many data paths if the structure involves cycles. Although a finite representation of the infinite set of data paths can be effectively constructed, in existing systems like Lorel, the problem is handled in a pragmatic way by considering only those paths which contain the same object at most once [AQM+97]. These *simple* data paths can be computed with FLORID by the following program:

```
[X,L,Y]:edge ← ssdb(X,L,Y), not X = Y.
[X,L,Y]:path[member↠{X,Y}] ← [X,L,Y]:edge.
[X,L,Y|P]:path[member↠{X,Y,Z}] ←
    [Y|P]:path[member↠{Z}], [X,L,Y]:edge, not [Y|P][member↠{X}].
```

Here we use Prolog's list notation $[X_1, \ldots, X_n|Xs]$, so the X_i are the first n elements and Xs is the tail of the list. The last rule involves non-stratified negation and yields the desired results when evaluated with FLORID's inflationary semantics.

[3] A shorthand notation, borrowed from XSQL [KKS92], is $x_0..\ell_1[X_1]..\ell_2[X_2]..\ \cdots\ ..\ell_n[X_n]$; cf. [FLU94].

3.2.2. General Path Expressions

Although simple path expressions augmented with the features presented in Section 2.2 in combination with rules already provide a powerful language for navigating on *ssd*, it is sometimes convenient to consider more general path expressions in the spirit of regular expressions. Below we consider an expressive language for *general path expressions* (*gpe*'s) and show that they can be specified and implemented concisely and elegantly in the DOOD framework of FLORID. Our *gpe*'s borrow from the general path expressions of Lorel [AQM+97], and are further extended with features for simplifying path expressions in the spirit of [VdBV93].

Definition 4 (General Path Expressions) Let $\mathcal{L}$ be a set of labels (strings). The set *GPE* of *general path expressions* is the least set such that:

- $\mathcal{L} \cup \{\mathsf{any}\} \subseteq GPE$. (A)
- If $M, N \in GPE$ and $n \in \mathbb{N}_0$, then the following are in *GPE*:
  $(M{\cdot}N)$, $(M|N)$, $(M)^*$, $(M)^+$, $(M)^?$, $(M)^{-1}$, $(M)^n$. (B)
- If φ is binary relation symbol, then $\mathsf{if}(\varphi) \in GPE$. (C)
- If $\ell \in \mathcal{L}$ and ψ is a unary relation symbol then $\mu(\ell), \mu(\ell, \psi) \in GPE$. (D)

The meaning of primitive *gpe*'s in (A) and regexp-like *gpe*'s in (B) should be clear. The intuition of the conditional label "$\mathsf{if}(\varphi)$" is as follows: given a binary predicate φ, navigation along $x \overset{M}{\twoheadrightarrow} y$ via $\mathsf{if}(\varphi)$ is allowed only if $\varphi(x, M)$ holds; the nodes reachable in this way are referenced by $x..\mathsf{if}(\varphi)$. For example, if $\varphi(x, M) :=$ "$\mathsf{substr}(M, x)$" then we may move from x to y along an edge labeled with M only if M is a substring of x (so x should also be a string). The basic idea of μ is to find the closest reachable node whose incoming edge is labeled with ℓ (see below).

For notational convenience, parentheses may be omitted whenever operator precedence is clear from the context.

Semantics of General Path Expressions

Let $E \in GPE$ be a general path expression. Again, we are interested in the object value of $x_0..E$, i.e., those nodes reachable from x_0 via a sequence $\ell_1, \ldots, \ell_n$ of labels $\ell_i \in \mathcal{L}$ such that $\ell_1, \ldots, \ell_n$ "matches" E. The meaning of *GPE* can be defined in an elegant way by viewing expressions $E \in GPE$ as *generalized labels*. Since every label ℓ can be seen as a relation $\ell := \{(x, y) \in \mathcal{N} \times \mathcal{N} \mid (x, \ell, y) \in \mathcal{D}\}$, every *gpe* also defines a binary relation on $\mathcal{D}$. These relations are defined by structural induction as illustrated in Figure 3.

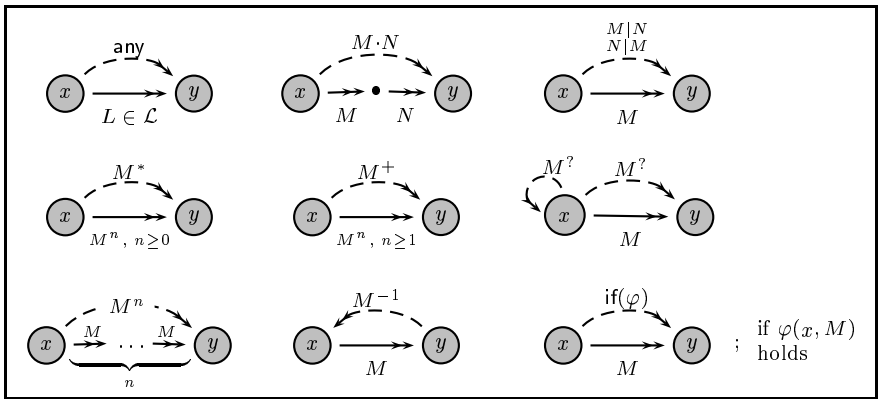


Fig. 3: Graphical Semantics of *gpe*'s

In addition to the operators specified in Figure 3, "any" matches any label $\ell \in \mathcal{L}$.

Example 8 (Navigation on the DBLP Skeleton) Assume we are interested in all url's reachable from the root of the DBLP skeleton, and which are referenced by a label containing the substrings "VLDB" *or* "SIGMOD". Using *gpe*'s this may be expressed in FLORID by the query

$$\text{?- dblp}\cdot\text{any}^*\cdot(\text{if}(\varphi_{vldb})|\text{if}(\varphi_{sigmod}))$$

where the predicates φ_{vldb} and φ_{sigmod} are true if the label contains the corresponding strings (e.g., $\varphi_{vldb}(\_, M)$ holds iff substr("VLDB",M) holds). See Figure 4 for the concrete FLORID syntax of *gpe*'s.

Finally, the expressions $\mu(\ell)$ and $\mu(\ell, \psi)$ use an idea from [VdBV93] for simplifying path expressions by finding the shortest paths ending in a given label: $x..\mu(\ell)$ yields those objects which are reachable from x via a sequence $\ell_1, \ldots, \ell_n, \ell$ such that there is no other sequence $\ell'_1, \ldots, \ell'_k, \ell$ with $k < n$. Thus, only the shortest sequences ending in ℓ are considered. $\mu(\ell, \psi)$ is a generalization which considers a data path $x_0, \ell_1, \ldots, x_n, \ell, x_{n+1}$ only if the penultimate object x_n satisfies a property $\psi(x_n)$. The motivation for using $\mu(\ell)$ and $\mu(\ell, \psi)$ in *ssdb*'s is that it will often match the "conceptually closest" nodes reachable via a certain label:

Example 9 (Contact Address) Consider the conceptual object-oriented schema depicted in Figure 1. Assume you want to know a contact address for a paper x (i.e., x:paper holds). This can be either specified as "if the address of one of the authors is known, take this address, else look for the address of one of the editors of the proceedings. If this address cannot be found, ask the conference series board ..." and so on. Or shorter: "*look for a label* address *which is closest to the paper*":

$$\text{?- } x..\mu(\text{address}) = \text{Y}.$$

Here, Y is bound to all result objects (i.e., addresses) having minimal distance from x.

This is also a standard situation when browsing through the Web: Assume you are on a Web page and you want to send an e-mail to the author: Just look for the nearest "mailto"-link.

3.2.3. General Path Expression in F-logic

General path expressions immediately lend themselves to an intuitive and elegant specification in F-logic using rules and simple path expressions. There are several ways to represent the operators of *GPE* in F-logic. Here, for notational simplicity, they are represented as id-terms:[4] For example, we denote $(M{\cdot}N)$ as "conc(M, N)" and $(M)^*$ as "star(M)", etc. Using this notation, Definition 4 can be directly translated into an equivalent F-logic program with path expressions as shown in Figure 4. The correctness of this translation is immediate for (A)–(C). The rules (D) defining $\mu(\ldots)$ make use of (non-stratified) negation in order to obtain the "nearest" possible result objects. It should be clear that by applying an *inflationary semantics* (as is the default in FLORID), the desired meaning is captured by the rules (D).[5] Finally, note that the rules defining if(φ) and $\mu(\ell, \psi)$ are in fact rule *schemas*, i.e., have to be instantiated with the corresponding predicate symbols φ (binary) and ψ (unary), respectively.

Bottom-Up Evaluation in FLORID

In order to ensure that P_{gpe} can be executed using standard bottom-up techniques as employed in FLORID, additional goals have to be added to the rules (B)–(D): The first reason is that some rules (e.g., for $X \overset{M|N}{\twoheadrightarrow} Y$) are not *range-restricted*, i.e., contain variables in the head which are not bound by a positive literal in the body. Moreover, even if rules *are* range-restricted they may define—due to the use of function symbols in the head—an infinite number of objects representing the generalized labels.[6] A simple cure to both problems is to constrain the rules by: (i) adding the

[4] Other more "object-oriented" encodings are possible, e.g., $M.conc@(N)$ for $M{\cdot}N$.

[5] ... since $\ell \in \mathcal{L}$. For nested applications like $\mu(M|\mu(N))$, other (more complex) rules have to be used.

[6] This mirrors the fact that *GPE* itself is infinite.

P_{gpe}:

(A)	X[L$\twoheadrightarrow$Y], X : node, Y : node, L : label $\leftarrow$ ssdb(X,L,Y).	% $X \overset{L}{\twoheadrightarrow} Y$
	X[any$\twoheadrightarrow$Y] $\leftarrow$ X[L$\twoheadrightarrow$Y], L : label.	% $X \overset{\mathsf{any}}{\twoheadrightarrow} Y$
(B)	X[conc(M,nil)$\twoheadrightarrow$Y] $\leftarrow$ X[M$\twoheadrightarrow$Y].	% $X \overset{M.\epsilon}{\twoheadrightarrow} Y$
	X[conc(M,N)$\twoheadrightarrow$Y] $\leftarrow$ X..M[N$\twoheadrightarrow$Y].	% $X \overset{M.N}{\twoheadrightarrow} Y$
	X[or(M,N)$\twoheadrightarrow$Y] $\leftarrow$ X[M$\twoheadrightarrow$Y].	% $X \overset{M\|N}{\twoheadrightarrow} Y$
	X[or(M,N)$\twoheadrightarrow$Y] $\leftarrow$ X[N$\twoheadrightarrow$Y].	% $X \overset{M\|N}{\twoheadrightarrow} Y$
	X[star(M)$\twoheadrightarrow$X] $\leftarrow$ X : node.	% $X \overset{M^*}{\twoheadrightarrow} X$
	X[star(M)$\twoheadrightarrow$Y] $\leftarrow$ X..star(M)[M$\twoheadrightarrow$Y].	% $X \overset{M^*}{\twoheadrightarrow} Y$
	X[plus(M)$\twoheadrightarrow$Y] $\leftarrow$ X[M$\twoheadrightarrow$Y].	% $X \overset{M^+}{\twoheadrightarrow} Y$
	X[plus(M)$\twoheadrightarrow$Y] $\leftarrow$ X..plus(M)[M$\twoheadrightarrow$Y].	% $X \overset{M^+}{\twoheadrightarrow} Y$
	X[opt(M)$\twoheadrightarrow$X] $\leftarrow$ X : node.	% $X \overset{M^?}{\twoheadrightarrow} X$
	X[opt(M)$\twoheadrightarrow$Y] $\leftarrow$ X[M$\twoheadrightarrow$Y].	% $X \overset{M^?}{\twoheadrightarrow} Y$
	X[inv(M)$\twoheadrightarrow$Y] $\leftarrow$ Y[M$\twoheadrightarrow$X].	% $X \overset{M^{-1}}{\twoheadrightarrow} Y$
	X[to(M,0)$\twoheadrightarrow$X] $\leftarrow$ X : node.	% $X \overset{M^0}{\twoheadrightarrow} X$
	X[to(M,K1)$\twoheadrightarrow$Y] $\leftarrow$ X..to(M,K)[M$\twoheadrightarrow$Y], K1 = K + 1.	% $X \overset{M^{k+1}}{\twoheadrightarrow} Y$
(C)	X[if(φ)$\twoheadrightarrow$Y] $\leftarrow$ X[M$\twoheadrightarrow$Y], φ(X,M).	% $X \overset{\mathsf{if}(\varphi)}{\twoheadrightarrow} Y$
(D)	X[mu(L)$\twoheadrightarrow$Y] $\leftarrow$ X..star(any)[L$\twoheadrightarrow$Y], $\neg$ X..mu(L).	% $X \overset{\mu(L)}{\twoheadrightarrow} Y$
	X[mu(L,ψ)$\twoheadrightarrow$Y] $\leftarrow$ ψ(X..star(any))[L$\twoheadrightarrow$Y], $\neg$ X..mu(L,ψ).	% $X \overset{\mu(L,\psi)}{\twoheadrightarrow} Y$

Fig. 4: General Path Expressions in F-logic

goal E : gpe to the body, for every rule with a head of the form $X[E\twoheadrightarrow Y]$, and (ii) constraining gpe to a finite set.

For example, by applying (i) to $M|N$ we obtain the rules

```
X[or(M,N)↠Y] ← or(M,N) : gpe, X[M↠Y].
X[or(M,N)↠Y] ← or(M,N) : gpe, X[N↠Y].
```

In this way, all rules (B)–(D) are modified. Requirement (ii) is accomplished by restricting the class gpe to those expressions which are relevant for answering a given *gpe* query (see Figure 5): Every expression E of class query is considered as an element of gpe. Program $P_{\downarrow}$ in Figure 5 computes the finitely many subexpressions which are relevant for a given query E. Assuming there is a class root containing all relevant root objects in the *ssdb*, all objects Y reachable from a root object X via the *gpe* E are retrieved by executing the query

```
?- E : query, X : root, X..E = Y.
```

which highlights once more the usefulness and elegance of employing a DOOD framework.

Although the program $P_{gpe} \cup P_{\downarrow}$ is already executable in FLORID, further optimizations are possible: To this end, instead of the simple class gpe, one has to define a predicate relevant(X, E) which computes from a given query $Q \in GPE$ only those intermediate objects X and *gpe*'s E applicable to X which are relevant for answering Q. Observe that such optimizations use essentially the same ideas as the well-known *Magic-Sets* rewriting technique [BMSU86] developed for deductive rules.

Alternatively, instead of implementing *gpe*'s via deductive rules, they could be implemented directly in the system, resulting in a more efficient solution. However, an important advantage

$P_\downarrow$:

```
conc(star(or(a,b)),mu(c)) : query.        % Example query:  ((a|b)*·μ(c))
query :: gpe.                             % queries are gpe's
-------------------------------------------------------------------------
% Compute downward closure of gpe:
label :: gpe.      any : gpe.             % L ∪ {any} ⊆ GPE
M : gpe, N : gpe ← conc(M,N) : gpe.       % M·N
M : gpe, N : gpe ← or(M,N) : gpe.         % M|N
M : gpe ← star(M) : gpe.                  % M*
M : gpe ← plus(M) : gpe.                  % M+
M : gpe ← opt(M) : gpe.                   % M?
M : gpe ← inv(M) : gpe.                   % M-1
M : gpe ← to(M,0) : gpe.                  % M0
to(M,K1) : gpe ← to(M,K) : gpe, K>0, K1=K–1.   % Mk
M : gpe, star(any) : gpe ← mu(M) : gpe.   % μ(M)
M : gpe, star(any) : gpe ← mu(M,_) : gpe. % μ(M, ψ)
```

Fig. 5: Downward Saturation of Query Expressions

of the deductive approach taken in FLORID is its flexibility and extensibility: new rule-defined methods can be added at any time without changing the system. For example, we may want to extend the rules of P_{gpe} with *path variables*[7], i.e., variables holding data paths. This can be accomplished by adding a parameter @(P) for data paths (represented as lists) to the methods in Figure 4 as follows:

```
X[L@([L,Y]) →→Y] ← ssdb(X,L,Y).
X[any@(P)→→Y] ← X[L@(P)→→Y], L : label.
X[conc(M,nil)@(P)→→Y] ← X[M@(P)→→Y].
X[conc(M,N)@(P)→→Y] ← X..M@(P1)[N@(P2)→→Y], append(P1,P2,P).
        ⋮              ⋮              ⋮
```

Note however that for a given *ssdb* of size n the number of data paths may be exponential in n (even in the absence of cycles), so the unrestricted use of path variables may be prohibitively expensive. In contrast, the original rules of P_{gpe} answer only *reachability* queries, i.e., the set of reachable objects without actually constructing data paths. In this case, the rules provide a **PTIME**-computable evaluation procedure for *gpe*'s:

Theorem 1 (Complexity of *GPE*)
Let $\mathcal{D}$ be a ssdb. For any node x_0 of $\mathcal{D}$ and every $E \in GPE$, $x_0..E$ can be evaluated in time polynomial in the size $|E| + |\mathcal{D}|$.

Proof. $P_\downarrow$ in Figure 5 (which depends on the given E : query) computes all proper subexpressions of E, which clearly is polynomial in $|E|$. In particular, there are only polynomially many new generalized labels M : gpe for which the rules of P_{gpe} apply. Since $P_{gpe} \cup P_\downarrow$ is evaluated under inflationary semantics, it follows that the model of $(P_{gpe} \cup P_\downarrow \cup \mathcal{D})$ is computable in polynomial time. □

4. QUERYING THE WEB

An important example for semistructured data as described in the previous section is the link structure (i.e., Web skeleton) of a set of Web documents. In this section, we describe an extension

[7] In contrast to the *object variables* $X_1, \ldots, X_n$ used in (***) in Section 3.2.1.

of FLORID for querying link structure *and* contents of Web documents: First, a DOOD model for Web documents is proposed and data-driven Web exploration using FLORID is introduced. We then present the declarative semantics of the Web extension based on F-logic.

4.1. The Web Model

Every resource available in the Web has a unique address, called *url* (*Uniform Resource Locator*). Typically, the document associated with an url contains hyperlinks to other url's which in turn refer to further Web documents. Thus, in F-logic, url's and Web documents can be modeled as shown in Figure 6 using two classes url and webdoc, respectively:

Every url is a string which provides a special single-valued method get (see below); the resulting object is a Web document, i.e., an instance of webdoc. Web documents are (conceivable as) strings which provide a parameterized multi-valued method hrefs@(*Label*), where *Label* is a string, yielding instances of url.[1] Moreover, the *method* url is applicable to Web documents and yields instances of *class* url (note the dual use of url, i.e., as method name and as class name).

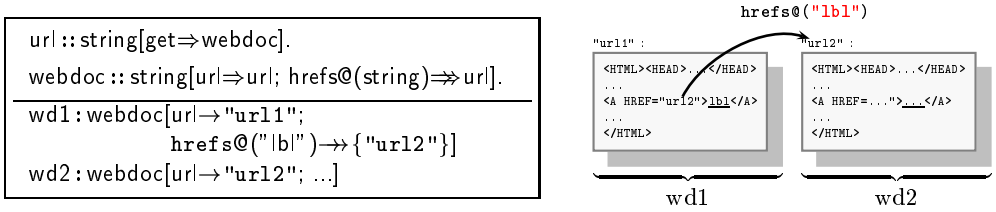


Fig. 6: F-logic Web Model: Signature and Example Data

When retrieving a Web document, certain *system-defined* methods are automatically filled in by the system; additionally, *user-defined* methods can be defined via rules. In FLORID, in addition to url and hrefs@(...), there are several other system-defined methods for Web documents, including the following:

webdoc[author⇒string; type⇒string; length⇒integer; modif⇒string; errors⇒⇒string; ...].

Here, the multi-valued method errors contains the reasons of the failure (e.g., *server does not exist*, *page not found*, *connection timed out*, ...) if the Web access fails. On successful access, the methods modif and type return the most recent modification time and the document type (HTML, ASCII, etc.), respectively.

Clearly, this schema reflects only some very basic properties of Web documents, without further exploiting the document type or structure. If necessary, the schema can be refined by declaring subclasses of webdoc, e.g., for BibTEX files, tables, etc. In particular, one may define a subclass sgmldoc such that the parse-tree of fetched SGML documents can be analyzed:[2]

sgmldoc :: webdoc[root⇒parsetree].

Here, instances of parsetree will have (usually parameterized) methods which can be used to navigate on the parse-trees.

Remark

Note that the Web skeleton in Example 6 considers only the *structural* aspect of Web documents and thus uses a simplified Web model, i.e., which corresponds to the signature url[string⇒⇒url]. In contrast, in order to query the *contents* of Web documents, we have to distinguish between the actual Web documents (instances of webdoc) and their url's (instances of url). Indeed, as will be shown in Example 11, the Web skeleton of a number of Web documents can be automatically extracted with FLORID.

[1] For a more detailed modeling, one could also include e.g. the byte-position of the link as a parameter, resulting in a single-valued method: webdoc[hrefs@(string,integer)⇒ url].

[2] This feature has recently been incorporated into FLORID.

4.2. Data-Driven Web Exploration

Although the Web is finite, for obvious reasons it is practically impossible to have access to the Web as a whole. The problem when integrating Web access into the F-logic framework lies in the necessity to load a beforehand unknown number of Web documents in order to extract some of their data which then has to be added to the logical model. In order to obtain a declarative semantics, loading Web documents is completely data-driven: new documents are explored depending on information and links found in already known documents.

The exploration strategy uses the semistructured data model given in Section 3, where nodes and labels are instances of the classes node and label, respectively: Exploration follows the *Web skeleton* whose nodes are url's and whose labeled edges are the corresponding hyperlinks. By repeatedly accessing Web documents, analyzing them, and following those hyperlinks which are expected to be relevant, the database is generated.

Exploration Cycle

Starting with a set of initial url's, data-driven Web exploration can be seen as repeated application of an operator *explore* which maps url's to new knowledge:

$$explore(u) := analyze \circ access(u) \ .$$

In every step, the "known" area of the Web is extended and additional knowledge is added to the database: A Web document is accessed using an already known url, then the document is loaded into the database; by analyzing it, information is extracted and new url's become known.

Example 10 (Exploring the DBPL Server) Figure 7 illustrates the exploration process for the DBLP server: Starting with the root url $\star$, its entry page dblp is accessed. By following the links (cf. the skeleton given in Figure 2), among others, the *Information Systems* page and the VLDB page are accessed. In the next step, the respective volumes or contents of the proceedings are explored. The fourth step then extends the knowledge with the authors' data.

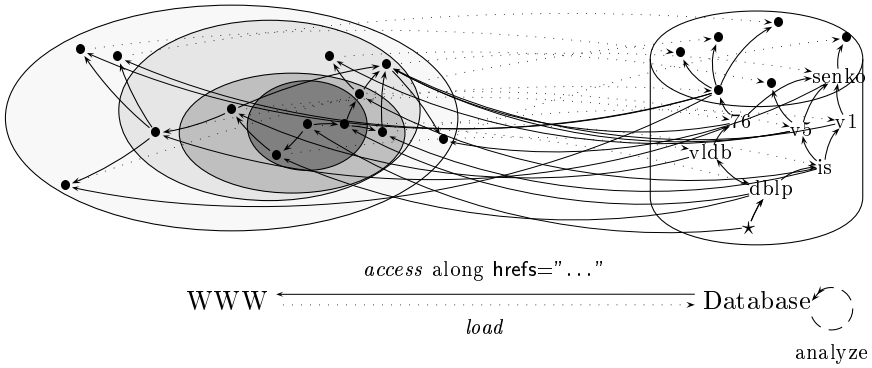


Fig. 7: Exploring the Web by Iteration of *access* and *analyze*

Web Exploration with FLORID

Apart from primitive functions for analyzing the contents of Web documents (e.g., for matching regular expressions), Web exploration involves accessing documents in the Web and adding them to the local F-logic database. This is accomplished in FLORID by calling the special method get for an instance u of the class url in the *head* of a rule, thereby creating the new oid "u.get" for the fetched Web document. After u.get has been created, system-defined methods are automatically "filled in" by FLORID, and the Web document, now named u.get, becomes an ordinary F-logic object whose contents may be conceived as a large string. In particular, u.get is "cached" and the url u is accessed only once. Note that the *potentially* system-defined methods for class webdoc are fixed in a given implementation—the *actually* system-defined (i.e., "filled") methods for u.get depend

on the result of accessing u. For example, if an error occurs, errors is defined and hrefs@(...) is undefined.

Extracting the Web Skeleton

As a typical and basic example, the Web skeleton can be automatically extracted with Florid using the generic skeleton extractor P_{ext} given in Figure 8:

P_{ext}:

```
root[src↠{u_1, ..., u_n}].                                  % (1)
node::url.                                                  % (2)
(U:node).get ← root[src↠{U}].                               % (3)
-------------------------------------------------------------------
Y:node, L:label, X[L↠{Y}] ←                                 % (4)
        X:node.get[hrefs@(L)↠{Y}], φ.
Y.get ← Y:node, ψ.                                          % (5)
```

Fig. 8: 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, the 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)↠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, for which ψ holds, are actually accessed and their contents retrieved in y.get.

Example 11 (DBLP Skeleton Extractor) A fragment of the DBLP server skeleton is retrieved (cf. Figure 2) 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 journal*)

After all relevant documents have been accessed, the query from Example 7 can be stated:

```
  ⋮
access :  http://www.informatik.uni-trier.de/~ley/db/journals/is/is22.html ...
process:  http://www.informatik.uni-trier.de/~ley/db/journals/is/is22.html (6588 bytes) ...
End evaluation
Answer to query :  ?- dblp.."Inf.  Systems"..L.."Michael E. Senko".
L/"Volume 5, 1980"
L/"Volume 1, 1975"
2 output(s) printed
```

It should be clear that there are many other ways of limiting the set of explored url's, which can be easily incorporated into P_{ext}. For example, for each node one may define a method depth such that source nodes have depth zero and a document referenced from another one with depth n has

itself depth $n+1$. Observe that this implies that depth must be defined as a multi-valued method, since each document may appear at different depths.[3]

4.3. Semantics of Web Access

In order to connect the model-theoretic semantics of F-logic programs with the abstract Web model explained above, the notion of a *Web interface* is defined:

Definition 5 (Web Interface) Let $\mathcal{URL}$ be the set of all url's and $\mathcal{R}$ a set of reserved names (0-ary functors). $\mathcal{R}$ contains the names for system-defined methods, here, at least the methods url, get, hrefs, and errors. $\mathcal{URL}$ and $\mathcal{R}$ are included into the set $\mathcal{F}$ of object constructors. Then, a *Web Interface* $\mathcal{W}$ is a tuple $(\mathcal{R}, explore)$, where

$$explore : \mathcal{URL} \to 2^{\mathcal{HB}^\star}$$

is a function mapping each $u \in \mathcal{URL}$ to a set of new facts (representing what is known after accessing and analyzing u).

Definition 6 (H-Web-Interpretation) An *H-Web-interpretation* wrt. a Web interface $\mathcal{W}$ is an H-interpretation $\mathcal{H} \subseteq \mathcal{HB}^\star$ that additionally satisfies the *Web access axiom*:

$$\text{for all } u \in U^\star, \text{ if } u : \mathsf{url}, u.\mathsf{get} \in \mathcal{H} \text{ then } explore(u) \subseteq \mathcal{H}.$$

Thus, if the reference u.get representing the document associated with the url u is active in an interpretation, then all information extracted by the analysis function *explore* is also contained in the interpretation. Given a program P, an H-Web-interpretation $\mathcal{H}$ with $\mathcal{H} \models P$ is called *H-Web-model* of P. Using these definitions it is easy to show [HLLS97]:

Theorem 2 *Every negation-free program P has a unique minimal H-Web-model wrt. $\mathcal{W}$.*

Bottom-up Evaluation

In deductive databases and logic programming, the evaluation of a program P can be characterized via the immediate consequences operator T_P: Beginning with the empty interpretation, the rules of the program are applied and the derived facts are added to the interpretation, until no new facts can be derived anymore. For negation-free programs, T_P is monotone, so the fixpoint $T_P^\omega(\emptyset)$ is the minimal model. For Web-F-logic programs P, an H-Web-interpretation has to satisfy the axioms given in Definition 2 and the Web access axiom. To ensure this, the T_P operator is extended as follows:[4]

Definition 7 For an F-logic program P and an H-interpretation $\mathcal{H}$,

$$\begin{aligned}
T_P^{\mathcal{W}}(\mathcal{H}) &:= \mathcal{H} \cup \{h \mid (h \leftarrow body) \in ground(P), \mathcal{H} \models body\}\ , \\
T_P^{\mathcal{W},0}(\mathcal{H}) &:= \mathcal{H}\ , \\
T_P^{\mathcal{W},i+1}(\mathcal{H}) &:= \mathcal{C}\ell(\ T_P^{\mathcal{W}}(T_P^{\mathcal{W},i}(\mathcal{H})) \cup \bigcup\{explore(u) \mid u : \mathsf{url},\ u.\mathsf{get} \in T_P^{\mathcal{W}}(T_P^{\mathcal{W},i}(\mathcal{H}))\}\ )
\end{aligned}$$

Theorem 3 *The fixpoint $T_P^{\mathcal{W},\omega}(\emptyset)$ is the minimal H-Web-model of a negation-free program P.*

Note, that due to the presence of functors and path constructors, the fixpoint may be infinite. FLORID's inflationary bottom-up evaluation is based on the above $T_P^{\mathcal{W}}$ operator; if *explore* is not used, the standard bottom-up evaluation is obtained.

[3] In principle, one could also use a single-valued method min_depth. However, as more and more parts of the Web are explored, the value of min_depth may have to be updated, which is not easily expressed in a declarative language like F-logic.

[4] Here, we use FLORID's inflationary semantics to also handle rules with negation.

4.4. Implementation Issues

FLORID does not work on H-interpretations but on semantic structures with a universe of oids: all information about an object is stored in an *object frame* which can be accessed by the oid of that object. Thus, for all references which adress the same object, there is only *one* oid and object frame.

The integration of Web access to the evaluation component simply amounts to a special treatment of the method get and the class url when adding new facts to the interpretation: Whenever the method get becomes defined for an object u,it is checked whether u is a member of the class url; and whenever a new instance is added to the class url, it is checked whether the method get is defined for it. In both cases, the function *explore* is called for u and the resulting facts are added to the interpretation. Web pages are loaded only when a *new* fact of this type is inserted. Thus, no document is fetched more than once. Note that the Web interface part does not interfere with program optimization techniques.

5. EXAMPLE: EXTRACTING AND RESTRUCTURING WEB DATA

In this section, a comprehensive example illustrates the concise and elegant programming style for querying and restructuring Web data with FLORID. Here, by restructuring we mean the reorganization of data extracted from the Web using logic rules. Again, the example is based on the DBLP server.

Apart from extracting the skeleton of a set of Web documents (i.e., their *link structure*) via the system-defined method hrefs@(...) (see Example 11), 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 and also in many other systems, is to view Web documents as (large) strings. Then, using *regular expressions*, patterns in Web documents can be easily exploited, e.g., to extract all strings between pairs of HTML tags like <h2> and </h2> (level-2 headings), or to analyze tables or lists. The regular expressions employed in FLORID include groups and format strings thereby providing an already quite expressive language: The predicate

match(Str, $RegEx$, Fmt, Res)

finds all strings in the input string Str which match the pattern given by the (GNU) 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,

```
?- match("Time heals all wounds", "\(.*\) heals \(.*\) \(.*\)", "\1 \3 \2 heels", X).
```

yields X="Time wounds all heels". Instead of Fmt and Res, also lists of format strings and result variables can be given. For example,

```
?- match("<li><a href="is22.html">Volume 22, 1997</a>",
         "Volume \([0-9]+\), \([0-9]+\)", ["\1","\2"], [V,Y]).
```

binds the volume, 22, to V and the year, 1997, to Y. Since for an url u, the reference u.get denotes the fetched Web document, u.get can be used as first argument to the match predicate.

In principle, by combining regular expressions with the power of recursive rules, arbitrary complex parsing tasks can be handled with FLORID. Fortunately, in many cases which we have encountered, there is no need to introduce recursion for parsing, and regular expressions of moderate size are sufficient to extract the desired data. In order to directly exploit the *document* structure of hypertext (or other structured) documents, their parse-trees could be made available to the user using the corresponding built-ins.

Populating the Publications Database

In Example 11, it is shown how to extract the skeleton of the DBLP server. In the sequel, we show how (part of) the object-oriented publications database in Figure 1 can be populated using the contents of the DBLP pages. We consider the following simplified sub-schema:

```
paper[authors ⇒⇒string; title ⇒string].
journal_p :: paper[in_vol⇒journal_vol].
journal_vol[volume ⇒integer; number ⇒⇒integer; year ⇒⇒integer; of ⇒journal].
journal[name ⇒string].
```

First, the url's of the DBLP server at Trier, Aachen, and a local (and incomplete) mirror are defined for the object dblp:

```
dblp[trier →"http://www.informatik.uni-trier.de/~ley/db/index.html";
     aachen →"http://sunsite.informatik.rwth-aachen.de/dblp/db/index.html";
     local →"file:/home/dbis/florid/dblp/db/index.html" ].
```

To allow for easy substitution of the data source, a generic name dblp.root is defined and the corresponding page is accessed:

```
dblp[root →dblp.trier].    % define which source to use
(dblp.root:url).get.                % get the root page
```

Now we may follow the link named "Journal" of the Web document dblp.root.get and retrieve the (unique) journal page:

```
dblp[the_journal_page →U] ← dblp.root.get[hrefs@("Journals") →→{U}].
(dblp.the_journal_page:url).get.
```

For each journal, we create a (logical) oid f(J,U) which depends on the name J and url U of the corresponding journal. For every journal, the method page is defined which yields its url; the last two goals select only links which lead to journals:

```
f(J,U):journal[name→J; page→U] ←
        dblp.the_journal_page.get[hrefs@(J) →→{U}],
        substr("db/journals",U), not U = dblp.the_journal_page.
```

Now we may ask FLORID to find all names of journals:

```
Answer to query :  ?- _:journal[name →J].

J/"(Office) Information Systems (TOIS)"
J/"AI Communications"
⋮
J/"Transactions on Software Engineering (TSE)"
J/"Wirtschaftsinformatik"
77 output(s) printed
```

Next we get all volumes of all journals (restricted to a class rel_journal). Here, for the first time, we use match with regular expressions to extract the number and year of the volume. Observe the concise rule notation obtained due to FLORID's path expressions and the use of match:

```
"Information Systems" : rel_journal.
(J.page:url).get ← J:journal[name→N:rel_journal].      % get journal pages
U.get:journal_vol[of →J; year →Y; volume →V] ←
        J:journal[name→N].page.get[hrefs@(Label) →→{U}],
        match(Label, "Volume \([0-9]+\), \([0-9]+\)", ["\1","\2"], [V,Y]).
```

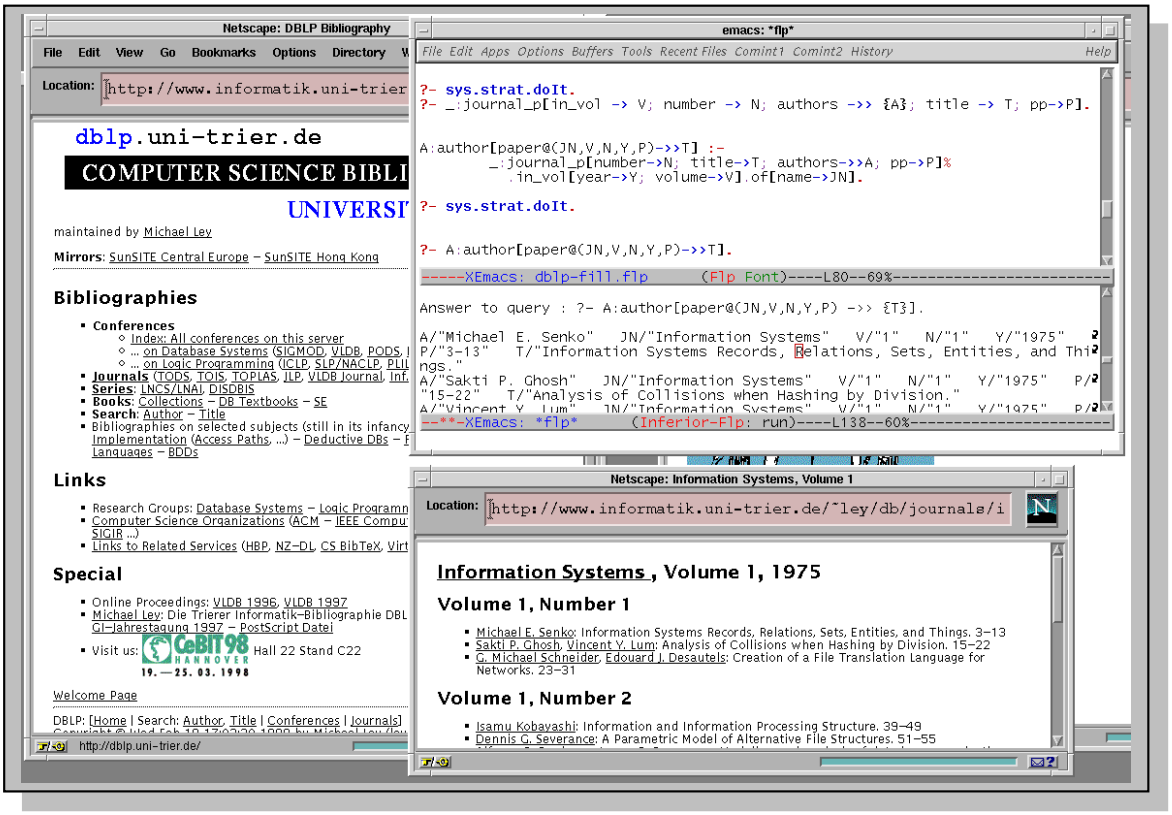


Fig. 9: Querying the DBLP Server with Florid

Finally, we are in position to extract the individual papers together with the volume, number, authors, title, and pages (pp) with the query given below. To this end, we make use of the textual structure of volume pages: Given a volume v, the contents of each number can be found between "Number" followed by the number ([0-9]+) and the next level-2 heading <h2> (see Figure 9). This text is matched by the first regexp in the query below. The authors of the volume are the names of the links which lead to url's in the /a-tree/, except the link to the authors index itself. For each paper, the HTML source contains three lines: the authors, the title, and the pages. Given an author, the last two subgoals extract this information.

```
f(V,T):journal_p[in_vol →V; number →N; authors →→{A}; title →T; pp→P] ←
        V:journal_vol,
        match(V, "Number \([0-9]+\)\(\([^<]\|<[^h]\|<h[^2]\|<h2[^>]\)+\)",
                 ["\1","\2"], [N,TS]),
        V[hrefs@(A) →→{U}], substr("/a-tree/",U), not A = "Author",
        strcat(A,".*\n\(.+\)\n\(.*\)",RegExp),
        match(TS,RegExp, ["\1","\2"],[T,P]).
```

Observe that we use the volume V[1] and the title T of a paper to create a unique f(V,T).

Restructuring Information

Assume we want to create the class of all authors of journal papers and define a parameterized method containing all relevant data about an author's journal papers. To do so, we simply have to query the data extracted so far and reorganize it as follows:

[1] The volume V is a Web document, not a number!

```
A:author[paper@(JN,V,N,Y,P)↠{T}] ←
        _:journal_p[number→N; title→T; authors↠{A}; pp→P]
                .in_vol[year→Y;volume→V].of[name→JN].
```

This rule exhibits again the power of path expressions for navigation on object-oriented databases. In FLORID's DOOD framework, also the computational power and elegance of deduction can be used: The following rules first extract the (irreflexive) co-author relation (restricted to the given journal papers) and then determine all "loners", i.e., authors who have no co-authors.

```
co_author(A,B) ← _:journal_p[authors↠{A,B}], not A=B.
loner(X) ← X:author, not co_author(X,_).
```

Using the transitive closure of the co-author relation, we may determine whether two authors are in different connected components:

```
tc_co_author(A,B) ← co_author(A,B).
tc_co_author(A,B) ← tc_co_author(A,C),tc_co_author(C,B).
?- sys.strat.dolt.
diff_component(A,B) ← A:author, B:author, not tc_co_author(A,B).
```

Here, the system query "?-sys.strat.dolt" prescribes that the rules below it are evaluated only after the rules above it have reached a fixpoint.

Finally, we can determine the number of different connected components by first collapsing all authors reachable from each other into one object and then counting the number of resulting equivalence classes:

```
f(A) = f(B) ← tc_co_author(A,B).
components(N) ← N=count{A; A=f(B)}.
```

6. CONCLUSION AND OUTLOOK

We have presented a deductive object-oriented perspective on management of semistructured data: DOOD languages are well-suited for exploring, modeling and (re)structuring semistructured data, due to (i) the semantical richness of the object-oriented model, and (ii) the concise, elegant, and expressive programming style provided by deductive rules. We have employed the DOOD language F-logic to show that both, navigation on semistructured data using general path expressions, and querying and restructuring of Web data can be handled in a unified formal framework. In particular, the use of a single general DOOD language supports rapid prototyping and adaptability to changing needs. The proposed framework is implemented in the FLORID system [FLO98]. So far, restructuring has been considered on the logical level only, i.e., using deductive rules, the extracted Web data is reorganized into different views. FLORID is currently extended with built-ins that will allow generation of XML [XML98] and HTML output so that restructuring is not only performed at the logical level but also at the document and presentation level, respectively.

As is the case with other existing Web query languages, Web pages are queried using regular expressions in FLORID. Although the combination of regular expressions and recursive rules already results in a very flexible language for data extraction, even more powerful parsing techniques for analyzing the document structure of arbitrary SGML documents will often simplify data extraction (e.g., access to list and table entries in HTML, or analysis of the tree structure of XML documents). To this end, a general SGML parser and a translator for mapping the parsed document structure into the corresponding F-logic structure have recently been incorporated into FLORID. Note that the formal framework easily accomodates arbitrary new document types, simply by extending the *analyze* function (Section 4) accordingly.

Acknowledgements — The authors thank the anonymous reviewers for many helpful comments.

REFERENCES

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

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[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.

[ARA98] ARANEUS Homepage. `http://poincare.inf.uniroma3.it:8080/Araneus/`, 1998.

[AV97] S. Abiteboul and V. Vianu. Queries and Computation on the Web. In *Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 262–275. Springer, 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)*, pp. 505–516, Montreal, Canada, 1996.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–15, Cambridge, Massachusetts, 1986.

[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.

[Cat94] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating Queries with Generalized Path Expressions. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 413–422, 1996.

[CRD94] Y. Chang, L. Raschid, and B. Dorr. Transforming Queries from a Relational Schema to an Equivalent Object Schema: A Prototype Based on F-Logic. In *International Symposium on Methodologies in Information Systems, (ISMIS)*, number 869 in LNCS, pp. 154–163. Springer, 1994.

[DBL98] DBLP Computer Science Bibliography. `http://www.informatik.uni-trier.de/~ley/db/`, 1998.

[FLO98] FLORID Homepage. `http://www.informatik.uni-freiburg.de/~dbis/florid/`, 1998.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 273–284, Santiago de Chile, 1994.

[FS98] M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Intl. Conference on Data Engineering (ICDE)*, pp. 14–23, Orlando, Florida, 1998.

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

[GPVdBVG94] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A Graph-Oriented Object Database Model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[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.

[HLLS98] R. Himmeröder, G. Lausen, B. Ludäscher, and C. Schlepphorst. FLORID: A DOOD-System for Querying the Web. In *Demonstration Session at EDBT*, Valencia, Spain, 1998.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In M. Stonebraker, editor, *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 393–403, 1992.

[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.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World-Wide Web. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 54–65, Zürich, Switzerland, 1995.

[LBT92] A. Lefebvre, P. Bernus, and R. Topor. Query Transformation for Accessing Heterogenous Databases. In *Post-JICSLP Workshop on Deductive Databases*, pp. 31–40, Washington, D.C., 1992.

[LSCS97] Z. Lacroix, A. Sahuguet, R. Chandrasekar, and B. Srinivas. A novel approach to querying the Web: Integrating Retrieval and Browsing. In *ER97, Workshop on Conceptual Modeling for Multimedia Information Seeking*, Los Angeles, 1997.

[LSS93] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the Logical Foundations of Schema Integration and Evolution in Heterogenous Database Systems. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, pp. 81–100, 1993.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *Proc. 6th Intl. Workshop on Research Issues in Data Engineering (RIDE)*, 1996.

[MLL97] W. May, B. Ludäscher, and G. Lausen. Well-Founded Semantics for Deductive Object-Oriented Database Languages. In Bry et al. [BRR97], pp. 320–336.

[MM97] A. Mendelzon and T. Milo. Formal Models of Web Queries. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 134–143, 1997.

[MMM96] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. of 5th International Conference on Parallel and Distributed Information Systems (PDIS'96)*, 1996.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 413–424, 1996.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Intl. Conference on Data Engineering (ICDE)*, pp. 132–141, 1996.

[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`.

[TSI98] TSIMMIS Homepage. `http://www-db.stanford.edu/tsimmis/tsimmis.html`, 1998.

[VdBV93] J. Van den Bussche and G. Vossen. An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 760 in LNCS, pp. 267–282. Springer, 1993.

[XML98] Extensible Markup Language (XML) 1.0. W3C recommendation, `http://www.w3.org/TR/REC-xml`, 1998.