Pl4Xml – An Swi–Prolog Library for Xml Data Management (Manual)

Dietmar Seipel
University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de

September 11, 2007

Abstract

Pl4Xml is a declarative Xml query, transformation and update language, which is implemented in and fully interleaved with the logic programming environment of Swi–Prolog.

We will introduce two Prolog document object models for Xml documents: the *field notation*, which represents Xml data as terms, and the *graph notation*, which represents Xml data as facts. Both simplify dealing with semi–structured data; instead of accessing a component by its argument position it becomes possible to access it by its attribute name, or to access nested components by a complex *path* or *tree expression*.

We will report about several practical *case studies* that we have done for extracting and transforming information from Xml documents in a declarative style.

Keywords.
Xml, semi–structured data, Prolog, query, update, and transformation languages, deductive databases

Contents

1 Introduction **3**

2 The Architecture of PL4XML **4**

3 XML Objects as PROLOG Terms **5**
- 3.1 XML Objects in Field Notation 5
- 3.2 Access to Components in Field Notation 7
- 3.3 Loading and Saving XML Documents 8

4 The Libraries FNPath and FNSelect **10**
- 4.1 Location, Assignment, and FN Trees 10
- 4.2 Evaluation of FNQuery Atoms in PROLOG 12
- 4.3 More General Location Steps 14
- 4.4 Related Concepts 15

5 XML Objects as PROLOG Facts **17**
- 5.1 XML Objects in Graph Notation 17
- 5.2 Additional Axes in Location Steps 19
- 5.3 Queries to the Graph Notation 20

6 The Library FNTransform **20**
- 6.1 FNTransform on Field Notation 20
- 6.2 FNTransform on Graph Notation 22

7 The Library FNUpdate **24**
- 7.1 Update 25
- 7.2 Insertion 28
- 7.3 Deletion 29

8 Case Studies **29**
- 8.1 Extracting Stock Data from Web Pages 30
- 8.2 Jean Paul 32

9 Conclusions and Future Work **34**

1 Introduction

XML is a well–known, self–describing data format, and it is often considered as the future of the World Wide Web. XML data can be considered as *semi–structured* data, since their schema or DTD might be unknown or frequently changing [19].

Query languages for object–oriented databases have been investigated since about 15 years, cf. the query language OQL of O_2 [2], the language Lorel [1], and the logic–based language *F–Logic* [11] and its successor languages. Since the emergence of XML the goal was to adapt these query languages to the processing XML data, cf. XQL (1998), XMLQL (1998), XQuery (also known as Quilt or XML Query), the extended style sheet language XSL (1999), and the pattern–based language XCERPT [6]. Many of the XML query languages including the emerging W3C standard XQuery [23] are based on the path language XPath (1999). Also specialized XML databases such as TAMINO have been developed. Recently, the goal of the *Semantic Web* [5] is to add a logic component with rules for *reasoning* about ontological XML data on the Web.

The logic programming language PROLOG can handle symbolic computations on complex, tree–structured objects nicely, and it can be used for reasoning about semi–structured data. In combination with the XML query, transformation and update language PL4XML, PROLOG is a powerful environment for handling XML–based applications. PL4XML integrates the XML transformation and update language with the query language into a single framework, such that an interleaved processing – including a combined backtracking – becomes possible.

Structure of the Manual

The rest of this manual is organized as follows: The *overall architecture* of PL4XML is described in Section 2. In Section 3 we will define the *field notation* for representing XML data as terms in a logic programming environment, and we will show how these data can be accessed nicely. In Section 4 the library FNPATH for processing data in field notation is introduced, and the semantics of FNPATH is described briefly. Section 5 defines the *graph notation* for representing XML data as PROLOG facts, as well as some further operations for accessing the data. The Sections 6 and 7 present the libraries FNTRANSFORM and FNUPDATE for transforming and updating, respectively, XML data. Some practical applications of PL4XML are reported in Section 8; in particular, it is shown how stock data can be extracted from HTML files, and how linguistic XML files can be enriched (annotated) by more semantical structure.

2 The Architecture of Pl4Xml

Pl4Xml is a Prolog library for querying, transforming, and updating Xml data, which is fully interleaved with the Prolog programming language. As such it combines the functionality of Xml tools such as XQuery, Xslt, and XUpdate with the programming language Prolog. The acronym Pl4Xml can be read as

- programming language for Xml or
- Prolog for Xml.

Pl4Xml has been used in several projects for processing Xml documents, but it can also simply be used as a data structure with access operations for complex objects in Prolog applications.

Document Object Models / Data Structures. Pl4Xml uses two alternative *representations* for Xml data in Prolog, the field notation and the graph notation.

Components. Pl4Xml consists of the two modules `fn_query` and `gn_query` within the unit `xml` of the DisLog Developers' Kit (DDK) [14]. The DDK is a comprehensive collection of Prolog libraries including disjunctive logic programming and deductive database features.

`fn_query` consists of the following components, which form the sub–language FnQuery.

- the path language FnPath,
- the selection language FnSelect,
- the transformation language FnTransform, and
- the update language FnUpdate.

The path language FnPath is used in all the other components. FnPath, FnSelect and FnTransform can operate both on field notation and on graph notation. FnUpdate – so far – can only operate on field notation. `gn_query` contains the predicates for converting between field notation and graph notation.

For handling basic data types such as strings and names various other Prolog predicates from the DDK are used, especially from the module `basic_algebra/basics`.

Usage. The Pl4Xml library can be used as an *application programming interface (API)* for processing Xml data within arbitray Prolog applications. Moreover, there exists a *graphical user interface (GUI)* for the rapid prototyping of small applications.

3 XML Objects as PROLOG Terms

Syntactically, both XML and PROLOG are based on nested *term structures*. We will show how XML structures can be represented and queried in PROLOG. Since XML can be used for representing complex objects, cf. [1], this gives us a way of handling complex objects in PROLOG or deductive databases.

Standard Term Representation of Complex Structures. The usual representation for relational or semi–structured data in PROLOG would be that the relation name becomes a predicate with the components of a tuple as its arguments. E.g., a relation `stock` with attributes `Index, Wkn, Date`, and `Value` can be represented by PROLOG facts of the form `stock(Index,Wkn,Date,Value)`. The schema information (i.e., the attribute names) is not stored in the PROLOG facts. There are the following implications of this representation: Firstly, the *selection* of attribute values is by argument position rather than by attribute, which is problematic if there are many arguments. Secondly, the update of nested term structures is complicated. E.g., the update of components within a structure `Stock`, such as the assignment `Stock.Value := 60`, is not possible without creating a new structure `Stock'`, because the *assignment* of variables is non–destructive in PROLOG.

3.1 XML Objects in Field Notation

Complex Objects in PROLOG. We will use *association lists* for representing lists of attribute/value pairs, cf. [1]. This data structure is familiar to LISP programmers. The following definition of association lists uses the infix operator "`:`" for pairing an attribute with its associated value.

Definition 3.1 (Association Lists)

1. If $\mathtt{a_i}$ and $\mathtt{v_i}$ $(1 \leq i \leq n)$ are PROLOG terms, then the $\mathtt{a_i} : \mathtt{v_i}$ are associations, and $[\mathtt{a_1} : \mathtt{v_1}, \ldots, \mathtt{a_n} : \mathtt{v_n}]$ is an association list.
2. Each $\mathtt{a_i}$ is called an attribute, and $\mathtt{v_i}$ is called the associated value.

For example, `[date:'14.12.2002', value:'30']` is an association list. Observe that association lists are PROLOG terms themselves; thus, it would be possible to have nested association lists, where some of the values $\mathtt{v_i}$ are association lists. E.g., `[f(1,a):7,b:[c:1]]` is an association list with the attributes $\mathtt{f}(1, \mathtt{a})$ and `b`, and with the corresponding values `7` and $[\mathtt{c} : 1]$, respectively. $[\mathtt{c} : 1]$ is itself an association list, wheras `7` is just a PROLOG constant.

A complex object with sub–objects that are selected by attributes $\mathtt{a_i}$, could be represented as an association list $[\mathtt{a_1} : \mathtt{O_1}, \ldots, \mathtt{a_n} : \mathtt{O_n}]$, where the $\mathtt{O_i}$, $1 \leq i \leq n$, are the association lists for the sub–objects. An object without any sub–objects would then be represented as a PROLOG constant.

Association lists have got several advantages when compared to ordinary PROLOG facts of the form $\mathtt{o}(\mathtt{O}_1, \ldots, \mathtt{O}_n)$: Firstly, the sequence of attribute/value pairs $\mathtt{a}_i : \mathtt{O}_i$ is arbitrary. Secondly, values O_i can be accessed by attributes $\mathtt{a}_i$ rather than by argument positions. Thirdly, the database schema can be changed at run time. Fourth, null values can be omitted, and new values can be added at run time.

In the following we will introduce a PROLOG representation – called field notation – of XML elements that represents the attribute/value pairs and the sub–elements as association lists, and we will show that XML elements in field notation can be queried and modified very elegantly.

XML **Objects in** PROLOG**.** The following XML example shows a database with information about the charts of the German stock index `dax100`; each chart has an identifying attribute `wkn` (identifier for stock shares; in German: Wertpapierkennummer) and entries with the attributes `date` and `value`:

```
<stocks index="dax100">
   <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
   <chart wkn="600800">
      <entry date="14.12.2002" value="40"/>
      <entry date="15.12.2002" value="50"/>
   </chart>
</stocks>
```

An XML element `<T` $a_1 =$ "v_1" $\ldots a_n =$ "v_n"`>` $\ldots$ `</T>` with the tag "`T`" can be represented as a PROLOG term `T:As:C`, where `As` $= [a_1 : v_1, \ldots, a_n : v_n]$ is an association list for the attribute/value pairs and `C` represents the contents (i.e., the sub–elements) of `O`. We call `T:As:C` an FN–triple. For example, the XML element above can be represented by the following FN–triple:

```
stocks:[index:dax100]:[
   chart:[wkn:200400]:[
      entry:[date:'14.12.2002', value:30]:[] ],
   chart:[wkn:600800]:[
   entry:[date:'14.12.2002', value:40]:[],
      entry:[date:'15.12.2002', value:50]:[] ] ]
```

Definition 3.2 (Field Notation)

1. If `T` and `C` are PROLOG terms and `As` is an association list, then the PROLOG term `O = T:As:C` is called an FN–triple with the tag `T` and the contents `C`.

- If As $= [a_1 : v_1, \ldots, a_n : v_n]$, then each a_i is called an attribute of O and v_i is called the corresponding value.
- If As $= [\,]$, then O can alternatively be represented as T : C.

2. Given an Fn–triple O = T:As:C, such that C $= [c_1, \ldots, c_m]$ is a list of Fn–triples, then each c_i is called a sub–element of O.

Observe, that the operator ":" is right–associative in Pl4Xml. Thus, an Fn–triple is perceived as a pair T : (As : C), and a list of Fn–triples can be considered as an association list with the selectors T.

The call fn_item_parse(I1, I2) normalizes the two forms of Fn–triples to the longer form: it transforms an abbreviated Fn–triple I1 = T:C to the long form I2 = T:[]:C, and it leaves T : As : C unchanged.

3.2 Access to Components in Field Notation

$Stocks. In the following we will use the abbreviation $Stocks$ for a frequently needed Fn–triple containing one stock chart with one entry, which is given by the following Xml element:

```
<stocks index="dax100">
   <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
</stocks>
```

Selection. Given an Fn–triple O and two Prolog terms A and X. The binary infix–predicate ":=" allows for accessing the sub–elements and the attribute values of O, which corresponds to the *child–axis* and the *attribute–axis*, respectively, of XPath.

- The call X := O/A computes all most general substitutions θ, such that Xθ is a sub–element with the tag Aθ of Oθ.
- Equivalently, it is possible to write X := O^A.
- The call X := O@A computes all most general substitutions θ, such that Xθ is the value of the attribute Aθ of Oθ.

This reminds of the evaluation of arithmetic expressions in Prolog, X is 3*(4+5), where "is" is an infix–predicate symbol with two arguments, which evaluates the arithmetic term $3*(4+5)$ and assigns the result to the first argument "X".

The predicate ":=" can handle *complex path expressions* for selecting sub–components. E.g., we get:

```
?- X := $Stocks/chart, Y := $Stocks@index,
   Z := $Stocks/chart@wkn.
```

```
X = chart:[wkn:200400]:[
       entry:[date:'14.12.2002', value:30]:[]],
Y = dax100,
Z = 200400
```

In pure PROLOG the selection `Z := $Stocks/chart@wkn` with a complex path expression would look much more complicated:

```
$Stocks = _:_:C,
member(chart:As:_, C),
member(wkn:Z, As).
```

It is also possible to use *variables* as selectors. Then, all possible selection paths (using child or attribute selectors) can be computed:

```
?- X := $Stocks/chart/S.

X = entry:[date:'14.12.2002', value:30]:[], S = entry ;
X = '14.12.2002', S = entry@date ;
X = 30, S = entry@value
```

Assignment. We can assign new values to attributes or elements. The following statement updates the `value` attributes of the `entry` elements of a selected `chart` element:

```
?- X := $Stocks/chart*[/entry@value:40].

X = chart:[wkn:200400]:[
       entry:[date:'14.12.2002', value:40]:[]].
```

`X` is obtained by first selecting the sub–element with the tag `chart`, and by then modifying the `value` attribute of the embedded `entry`–element. Observe, that we have to create a new object `X` holding the result, since in PROLOG there exists no destructive assignment.

The following statement inserts a new sub–element into the selected `chart` element:

```
?- X := $Stocks/chart*[
      /entry:[date:'17.12.2002', value:60]:[]].

X = chart:[wkn:200400]:[
       entry:[date:'14.12.2002', value:30]:[],
       entry:[date:'17.12.2002', value:60]:[]]
```

3.3 Loading and Saving XML Documents

We use the SGML/XML parser *sgml2pl* of SWI–PROLOG [22] for loading XML documents. *sgml2pl* represents XML elements as terms of the form

element(`Tag, Attributes, Content`),

where `Tag` is the tag of the element, `Attributes` is the list of attribute/value assignments `A=V`, and `Content` is the list of sub–elements in *sgml2pl* representation. In Pl4Xml, we shorten this to the term `Tag:As:C`, where `As` is the corresponding list of attribute/value pairs `A:V` and `C` is the list of sub–elements in field notation, which corresponds to `Content`; I.e., we change "`=`" to "`:`" and we leave out the functor "`element`"; all terms of the *sgml2pl* representation with other functors are left unchanged.

Loading. We can load an Xml document into the system using the predicate `dread/3`. The call

```
?- dread(xml, 'stocks.xml', Stocks_2).

Stocks_2 = [
   stocks:[index:dax100]:[
      chart:[wkn:200400]:[...],
      chart:[wkn:600800]:[...] ] ]
```

reads the file `stocks.xml` into an FN–list consisting of exactly one Fn–triple.

If the file `stocks.xml` contains processing instructions or more than one root element (the latter is allowed in FnQuery, although is is not allowed in Xml), then the result argument of `dread/3` is a list containing these items.

Saving. We can save an Fn–triple as an Xml document using the predicates `dwrite`/2 and `dwrite`/3. The call

```
?- dwrite(xml, $Stocks).
```

writes the Fn–triple `$Stocks` to the console, and the call

```
?- dwrite(xml, 'stocks_2.xml', $Stocks).
```

writes it into the Xml file `stocks_2.xml`; the file is started by the Xml prolog `<?xml version='1.0' encoding='ISO-8859-1' ?>`. The predicate `dwrite/2` does not print this prolog.

Observe, that – unlike `dread`, which returns an Fn–list – `dwrite` usually will be called with an Fn–triple. Calling `dwrite` with an Fn–list writes the Fn–triples embraced by the tags `<>` and `</>`:

```
?- dwrite(xml, [a:[b:1]:[], c:[]]).
<>
   <a b="1"/>
   <c/>
</>
```

The same is obtained from the call `dwrite(xml, '':[a:[b:1]:[], c:[]])`.

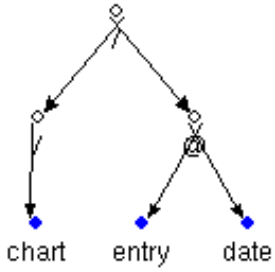

Figure 1: A Location Path in PROLOG

4 The Libraries FNPath and FNSelect

In this section we will describe the *basic* methods of FNSelect for the selection of sub–elements and the update of FN–triples using complex location expressions, which we call location trees and assignment trees in FNPath. More *advanced* access methods, such as the selection/deletion of all elements/attributes of a certain pattern, the transformation of sub–elements according to substitution rules (FNTransform, cf. Section 5), and the update (insertion, deletion) of sub–elements (FNUpdate, cf. Section 6) will be described in further sections.

4.1 Location, Assignment, and FN Trees

Location paths and location trees are defined inductively as terms over PROLOG terms and some other characters.

Definition 4.1 (Location Paths and Location Trees)

1. If `t` is a PROLOG term, then `/t`, `^t`, and `@t` are location steps.
2. If π_i $(1 \leq i \leq n)$ are location steps, then $\pi_1 \ldots \pi_n$ is a location path.
3. Every location path is a location tree. If π is a location path and τ_i $(1 \leq i \leq n)$ are location trees, then π `-[`$\tau_1, \ldots, \tau_n$`]` is a location tree.

For example, `/chart-[@wkn,/entry-[@date,@value]]` is a location tree. If `O` is an FN–triple and τ is a location tree, then `O` τ can be considered as a PROLOG term. In a location path – depending on their position – the functors /, ^, and @ are either perceived as right–associative, binary infix functors or as unary prefix functors with suitable precedences and types (cf. Section 4.2). E.g., the location path `/chart/entry@date` is parsed into the PROLOG term shown in Figure 1.

A location path is applied to an FN–triple `O` by iteratively applying the location steps, and a location tree π `-[`$\tau_1, \ldots, \tau_n$`]` is applied by first applying

π to `O` and by then applying the location trees τ_i in parallel. Thus, a query with *multiple location paths* returns a list of objects, namely one object for each selector:

```
?- X := $Stocks/chart-[@wkn, /entry-[@date, @value]].

X = [200400, ['14.12.2002', 30]].
```

If a path contains *variable symbols*, then all instances of the path which are an allowed path in the query object can be generated on backtracking:

```
?- X := $Stocks/chart/entry@Path.

X = '14.12.2002', Path = date ;
X = 30, Path = value

?- 30 := $Stocks/Path.

Path = chart/entry@value
```

If a location tree τ is not *compatible* with an Fn–triple `O`, then `O` τ does not return any results. E.g., τ =`/entry` is not compatible with our complex object, since `O` does not contain any (direct) sub–elements with the tag `entry`. Obviously, a path π containing two subsequent functors `@`, such as, e.g., `@a@b`, is not compatible with Xml objects `O` in field notation, since Xml attributes cannot have a complex structure.

Definition 4.2 (Association Trees, Assignment Trees)
Given a location path π_0.

1. If I is an Fn–triple, then π_0/I is an association path.
2. If `A : V` is an association, then π_0 `@ A : V` is an association path.
3. Every association path is also an association tree.
   If all τ_i $(1 \leq i \leq n)$ are association trees, then
   - π_0 `-` $[\tau_1, \ldots, \tau_n]$ is an association tree, and
   - π_0 `*` $[\tau_1, \ldots, \tau_n]$ is an assignment tree.

An assignment tree $\pi_0 * [\tau_1, \ldots, \tau_n]$ selects a sub–element of an Fn–triple `O` using π_0, and then it iteratively applies the association trees τ_i to change certain attributes or elements. It is possible to change components at arbitrary depth in the document.

```
?- X := $Stocks/chart*[/entry@value:40].

X = chart:[wkn:200400]:[
       entry:[date:'14.12.2002', value:40]:[]].
```

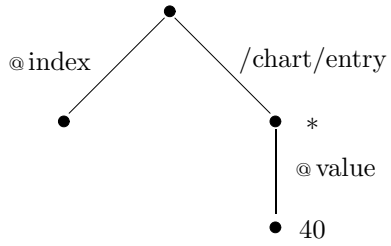

Figure 2: An FN–Tree

Location trees and assignment trees can be combined within *FN–trees*, which form the basis for FnQuery atoms.

Definition 4.3 (FN–Trees, FN–Query Atoms)

1. Both location trees and assignment trees are FN–trees. If π_0 is a path and τ_i $(1 \leq i \leq n)$ are FN–trees, then π_0 `-[`$\tau_1, \ldots, \tau_n$`]` is an FN–tree.
2. If `O` is an Fn–triple, τ is an FN–tree, and X is a Prolog term, then `O` τ is an FnQuery term and `X :=` `O` τ is an FnQuery atom.

An FN–tree represents a *labelled tree*, where each edge is labelled by a path. The nodes can be labelled by $*$ or by a Prolog term. Each path from the root to a leaf contains at most one node that is labelled by $*$; in this case the leaf is labelled by a Prolog term. The FN–tree that is used in the following example is given in Figure 2:

```
?- Y := $Stocks-[@index, /chart/entry*[@value:40]].

Y = [dax100, entry:[date:'14.12.2002',value:40]:[]].
```

4.2 Evaluation of FnQuery Atoms in Prolog

We have implemented a Prolog engine for evaluating FnQuery atoms, cf. [14]. The operators /, ^, and @ are defined as *right–associative*, binary infix operators with the same precedence, and at the same time as unary prefix operators using the call `op(Precedence,Type,Name)` for the following parameters:

Name	Precedence	Type
/	200	xfy
/	100	fx
^	200	xfy
^	100	fx
@	200	xfy
@	100	fx

Simplified Notations. A location tree or an association tree π `-` `[`$\circ\tau_1, \ldots, \circ\tau_n$`]` where all $\circ\tau_i$ start with the same selector $\circ \in \{ /, @ \}$, can equivalently be represented as $\pi \circ$ `[`$\tau_1, \ldots, \tau_n$`]`. An assignment tree $\pi *$ `[`$/\tau_1, \ldots, /\tau_n$`]` where all association trees start with /, is represented as $\pi *$ `[`$\tau_1, \ldots, \tau_n$`]`.

For example, `O-[/a,/c]` can be represented as `O/[a,c]`, and `O*[/a:5,/c:6]` can be represented as `O*[a:5,c:6]`.

`$Stocks_2`. In the following we will use the abbreviation `$Stocks_2$` for a frequently needed FN–triple containing two stock charts, which is given by the following XML element:

```
<stocks index="dax100">
   <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
   <chart wkn="600800">
      <entry date="14.12.2002" value="40"/>
      <entry date="15.12.2002" value="50"/>
   </chart>
</stocks>
```

Aggregation Operators and Embedded Computations. We can handle query terms `X` := `O` − `agg`($\mathtt{X}_1, \ldots, \mathtt{X}_n, \pi$) with *aggregation goals* with parameters $\mathtt{X}_1, \ldots, \mathtt{X}_n$, such as `O-average(/a)`, `O-sum(/a)`, and `O-nth(2, /a)`. Here, the path π is first applied to the object `O` by calling `Y` := `O` π; we find all results by `findall( Y, Y := O` π, `Ys )`. Then the aggregate `X` is obtained from the list `Ys` of all results `Y` of this call by calling `agg`($\mathtt{X}_1, \ldots, \mathtt{X}_n, \mathtt{Ys}, \mathtt{X}$). Arbitrary user–defined PROLOG predicates can be used for `agg`. For example, the query `X := O-agg(/a)` returns the list of all `a` sub–elements of `O`, if `agg` is defined by the fact `agg`(`Ys`, `Ys`).

The following query returns the average of all `value`–attributes within the second `chart` element:

```
?- X := $Stocks_2-nth(2, /chart)-average(/entry@value).

X = 45
```

First, the second `chart` element is selected; then the average over all `value` attributes of its `entry` elements is computed.

Finally, arbitrary *embedded computations* – such as computing the changes from one day to the next day by a user–defined function `changes` – can be started while evaluating a query term using the connnective "`--`":

```
?- X := $Stocks_2--changes, dwrite(xml, X).

<stocks index="dax100">
```

```
    <chart wkn="200400"/>
    <chart wkn="600800">
       <entry from="14.12.2002" to="15.12.2002"
          change="+25%"/>
    </chart>
</stocks>
```

4.3 More General Location Steps

A location path is a sequence $\pi = /s_1/s_2/\ldots/s_n$ of location steps s_i. In general a location step in PL4XML can look as follows:

```
Axis::Node_Test::Predicates
Axis::Node_Test
```

The pair `Axis::Node_Test` is used for selecting elements or attributes, and the list `Predicates` of conditions is used for testing the selected object. Analogously to XPath, the are suitable abbreviations for the axes `child` and `attribute` in a path: `/child::T` can be abbreviated to `/T`, and `/attribute::A` can be abbreviated to `@A`.[1] For location trees and assigment trees, the notations in the general from will be given later in this section. The general form can be used within the earlier form. Finally, note that it is always possible to write "^" instead of "/".

For example, the following call selects the `date` of all entries with a value greater than `45` (`Predicates = [@value=V, V > 45]` in the first location step) from the chart with the `wkn` given by `600800` (`Predicates = [@wkn=600800]` in the second location step):

```
?- X := $Stocks_2/chart::[@wkn=600800]
      /entry::[@value=V, V > 45]@date.

X = '15.12.2002',
V = 50
```

The FN–triple `$Stocks_2` has been defined in Section 4.2. As a side effect, also the corresponding values `V` are returned.

Axes and Node Tests. In general the following pairs are possible in PL4XML: `T`, `A`, and `N` can be any PROLOG term; in reasonable situations, `T`, `A`, and `N` will be variables, `'*'`, or a tag name for `T`, an attribute for `A`, and a number for `N`, respectively. `'*'` stands for a wild card variable `_`.

- `child::T` selects a child element with the tag `T`;

[1] Earlier, `/T` and `@A` have been called location steps, although they are abbreviations for the separator / together with a location step s_i.

- `attribute::A` selects the value of an attribute `A`;
- `tag::'*'` selects the tag name of the considered FN–triple;
- `attributes::'*'` selects the attributes of the considered FN–triple;
- `content::'*'` selects the content of the considered FN–triple;
- `nth_child::N` selects the `N`–th child element of the considered FN–triple;
- `self::'*'` selects the considered FN–triple itself;
- `descendent::T` selects a descendent element of the considered FN–triple with the tag `T`;
- `descendent_or_self::T` selects a descendent element of the considered FN–triple the tag `T`, or the FN–triple itself, if it has the tag `T`.

The backward axes, such as `parent`, are not possible for field notation, since an FN–triple does not have a reference to its parent. In Section 5, we will introduce another representation of XML, called graph notation, which will allow all axes of XPath.

The following example selects some components of a `stock` element:

```
?- Chart := $Stocks_2/chart::[@wkn=600800],
   T := Chart/tag::'*',
   As := Chart/attributes::'*',
   C := Chart/content::'*',
   Child := Chart/nth_child::2,
   Self := Chart/self::'*'.

Chart = chart:[wkn:600800]:[...],
T = chart,
As = [wkn:600800],
C = [ entry:[date:'14.12.2002', value:40]:[],
      entry:[date:'15.12.2002', value:50]:[] ],
Child = entry:[date:'15.12.2002', value:50]:[],
Self = chart:[wkn:600800]:[...]
```

Further Long Forms. A location or association tree π_0`--[`$\tau_1, \ldots, \tau_n$`]` can be written as π_0`/branch::[`$\tau_1, \ldots, \tau_n$`]`, and an assignment tree π_0`*[`$\tau_1, \ldots, \tau_n$`]` as π_0`/update::[`$\tau_1, \ldots, \tau_n$`]`.

4.4 Related Concepts

We will briefly relate FNPath to two representative concepts for dealing with XML data by an example: XQuery and F–Logic. FNPath differs from XQuery

and from F–Logic in the fact that FnPath is fully interleaved with Prolog, in the built–predicates, which are allowed in FN–trees, and in the library of built–in predicates, which we have implemented.

Xml **Query.** The following Prolog rule uses the features of FnPath for computing all stock values for a given day `Date`:

```
stock(Stocks, Date, entries:[]:Entries) :-
   findall( entry:[wkn:Wkn, value:Value]:[],
      ( Chart := Stocks/chart,
        Wkn   := Chart@wkn,
        Entry := Chart/entry,
        Date  := Entry@date,
        Value := Entry@value ),
      Entries ).
```

It can be applied to the Fn–triple `Stocks` given in Section 3.1. The corresponding `FLWR`–expression in Xml Query would look like follows:

```
<entries>
   { for $Chart in document("http://www.stocks.de")/stocks/chart
     let $Wkn = $Chart@wkn
     for $Entry in $Chart/entry
     let $Value = $Entry@value
     where Date = $Entry@date
     return <entry wkn="$Wkn" value="$Value"/> }
</entries>
```

F–Logic. F–Logic is also based on complex, nested structures, such as

```
stocks[ index -> dax100,
   charts --> {
      [ wkn -> 200400,
        entries --> {
           [ date -> '14.12.2002', value -> 30 ] } ],
      [ wkn -> 600800,
        entries --> {
           [ date -> '14.12.2002', value -> 40 ],
           [ date -> '15.12.2002', value -> 50 ] } ] } ]
```

which are called molecules; but a complex object is usually represented by a collection of molecules. To a large degree the access to components is comparable to FnPath, but F–Logic does not consider quite as powerful transformation facilities.

5 XML Objects as PROLOG Facts

If we store XML objects as PROLOG facts using an object/relational mapping, then we can implement the backward axes of XQuery as well.

5.1 XML Objects in Graph Notation

The graph notation is a relational representation of XML that is following and refining a proposal presented in [1], which is based on two relations `ref/3` and `val/2`.

The GN Database. The GN database can store many XML elements at the same time. We refer to the elements I and their descendent elements J by unique identifiers; $father(J)$ denotes the father of J in I, $tag(J)$ is the tag of J, and $text(J)$ is the text of a text element J.

- I is mapped to a fact

  `reference`(I^*, $tag(I)$, I, N),

  where I^* is an identifier that is not the id of any other element in the database.

- A non–text element J is mapped to a fact

  `reference`($father(J)$, $tag(J)$, J, N).

- For every attribute/value–pair $A : V$ of J we get a fact

  `attribute`(J, A, V).

- A text element J is mapped to a fact

  `value`($father(J)$, $text(J)$, N).

The order of the sub–elements K of a non–text element J is reflected by the numbers N in the facts `reference`(J, _, K, N) and `value`(J, $text(K)$, N), which must all be different. Thus, the stored XML elements can be reconstructed from the GN database.

Switching to Graph Notation. We can switch to graph notation by setting the DISLOG variable `fn_mode` to `gn`:

```
?- dislog_variable_set(fn_mode, gn).

Yes
```

The default for `fn_mode` is `fn` for field notation, which is active when the DDK is started.

Conversion between Field and Graph Notation. An FN–triple in field notation is stored into the GN database using the predicate `fn_to_gn/2`:

```
?- Item = chart:[wkn:600800]:[
      entry:[value:40]:['Mo', b:['Tue']],
      entry:[value:50]:['We'] ],
   fn_to_gn(Item, Id).

Id = '&2'
```

The returned identifier `'&2'` refers to the stored FN–triple. After that the database contains 8 facts, which can be listed by `gn_database_listing/0`:

```
?- gn_database_listing.

reference('&1', chart, '&2', 1).
reference('&2', entry, '&3', 2).
reference('&3', b, '&4', 4).
reference('&2', entry, '&5', 6).

attribute('&2', wkn, 600800).
attribute('&3', value, 40).
attribute('&5', value, 50).

value('&3', 'Mo', 3).
value('&4', 'Tue', 5).
value('&5', 'We', 7).
```

The stored FN–triple can be reconstructed from its identifier using `gn_to_fn/2`:

```
?- gn_to_fn('&2', Item).

Item = chart:[wkn:600800]:[
   entry:[value:40]:['Mo', b:['Tue']],
   entry:[value:50]:['We']]
```

The sub–elements of the `entry` element `'&3'` are reconstructed from the facts `value('&3', 'Mo', 3)` and `reference('&3', b, '&4', 4)`, and they are ordered according to the numbers in the last arguments.

Shallow parsing of `'&2'` yields an FN–triple whose sub–elements are given by their identifiers:

```
?- fn_item_parse('&2', Item).

Item = chart:[wkn:600800]:['&3', '&4']

Yes
```

Loading and Saving the GN Database. If the `fn_mode` is `gn`, then an item in graph notation which is referenced by `Id` can be loaded and saved from and to a file `F`, respectively, using the calls

- `dread(xml, F, [Id])` and
- `dwrite(xml, F, Id)`.

E.g., the XML file `stocks.xml` can be read into graph notation like follows:

```
?- dread(xml, 'stocks.xml', [Id]),
   dwrite(xml, user, Id).

<stocks index="dax100">
   <chart wkn="200400"> ... </chart>
   <chart wkn="600800"> ... </chart>
</stocks>

Id = '&2'
Yes
```

5.2 Additional Axes in Location Steps

The following additional location steps with backward axes are possible in GNQuery on graph notation: `T` can be any PROLOG term; in reasonable situations, `T` will be a variable, `'*'`, or a tag name. `'*'` stands for a wild card variable `_`.

- `parent::T` selects the parent element, if it has the tag `T`;
- `ancestor::T` selects an ancestor element with the tag `T`;
- `ancestor_or_self::T` selects an ancestor element of the considered FN–triple with the tag `T` or the FN–triple itself, if its tag is `T`;
- `following_sibling::T` selects the following sibling, if it has the tag `T`;
- `preceding_sibling::T` selects the preceding sibling, if it has the tag `T`;
- `following::T` selects an element with the tag `T` that is following the considered FN–triple in the document order, except the descendents;
- `preceding::T` selects an element with the tag `T` that is preceding the considered FN–triple in the document order, except the ancestors.

If `T` is a variable, then the test for the tag name always succeeds, and the tag name of the selected element is returned as the value of `T` (unless `T` is "`_`").

5.3 Queries to the Graph Notation

In graph notation all axes can be used that have been described for field notation before, and they have the same meaning.

```
?- X := '&2'/entry,
   Xs := '&2'/[ tag::'*', attributes::'*', content::'*',
      nth_child::'*', self::'*' ].

X = '&3',
Xs = [chart, [wkn:600800], ['&3', '&5'], '&3', '&2']
```

Additionally, the backwards axes, which are all based on `parent`, can be used.

```
?- Par := '&5'/parent::'*',
   Sib := '&5'/preceding_sibling::T1,
   Fol := '&4'/following::T2.

Par = '&2',
Sib = '&3', T1 = entry,
Fol = '&5', T2 = entry
```

The `b` element `'&4'` has two ancestors:

```
?- Anc := '&4'/ancestor::T.

Anc = '&3', T = entry ;
Anc = '&2', T = chart
```

6 The Library FnTransform

FnTransform can operate both on field and on graph notation. We have developed a *grammar rule* formalism similar to, but more powerful than, Xslt.

The available transformation methods are summarized in the call

```
fn_transform(Options, +X, ?Y)
```

for various forms of options and data `X,Y`. The data can be given as items in field or graph notation, or as files.

6.1 FnTransform on Field Notation

Transformations with Arbitrary Predicates. If `Predicate` is a binary predicate or a goal `P(A_1,...,A_n)` for a predicate `P` of the arity $n + 2$, then the calls

```
fn_transform([item, predicate(Predicate)], +Item_1, -Item_2)
fn_transform([file, predicate(Predicate)], +File_1, +File_2)
```

recursively transform XML elements; the transformation starts with the leaf elements. The first call first transforms the sub–elements `I1` of `Item_1` to form new elements `I3`, and then it calls `apply(Predicate, [I3, I2])` to form the sub–elements `I2` of the new FN–triple `Item_2`. The second call does the same for files.

E.g., consider the binary predicate `tr/2`: the first rule deletes the sub–elements of `chart` elements `chart:As_1:Es` and adds the number of these sub–elements to the attribute list `As_1` to create a new element `chart:As_2:[]` without sub–elements; the second rule changes the tag `stock` to `stock_summary`; the third rule is applied for all other elements, and it leaves them unchanged:

```
tr(chart:As_1:Es, chart:As_2:[]) :-
   length(Es, N),
   append(As_1, [entries:N], As_2).
tr(stocks:As:Es, stock_summary:As:Es).
tr(X, X).
```

We can apply `tr/2` to the stock data:

```
?- dread(xml, 'stocks.xml', [Item_1]),
   fn_item_transform(tr, Item_1, Item_2),
   dwrite(xml, Item_2).

<stock_summary index="dax100">
   <chart wkn="200400" entries="1"/>
   <chart wkn="600800" entries="2"/>
</stock_summary>
```

Transformations with Field Notation Grammars. A field notation grammar (FNG) is defined by rules for the binary, infix predicate `--->/2`. Given a file `File_S` defining `--->/2`. Then the calls

```
fn_transform([item, fng(File_S)], +Item_1, -Item_2)
fn_transform([file, fng(File_S)], +File_1, +File_2)
```

apply `--->/2` to an FN–triple or an XML file to create a new FN–triple or XML file, respectively. Alternatively, we can write the following for the first call:

```
Item_2 := Item_1/transform::File_S
```

E.g., consider the FNG defined in the file `stock.fng` with the same functionality as the predicate `tr/2` from above:

```
chart:As_1:Es ---> chart:As_2:[] :-
   length(Es, N),
   append(As_1, [entries:N], As_2).
stocks:As:Es ---> stock_summary:As:Es.
X ---> X.
```

Then the following call transforms the XML file `stocks.xml` into another XML file `stocks_2.xml`:

```
?- fn_transform_xml_file_fng(
      'stock.fng', 'stocks.xml', 'stocks_2.xml').

Yes
```

The new XML file `stocks_2.xml` looks as follows:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<stock_summary index="dax100">
   <chart wkn="200400" entries="1"/>
   <chart wkn="600800" entries="2"/>
</stock_summary>
```

Deleting Elements with Field Notation Grammars. An FNG rule of the form "`T:As:Es ---> ''`" will delete all elements with the tag `T` from the XML document. In fact, these elements are replaced by "`''`".

Transformations in Location Tree Expressions. For embedding a grammar rule transformation of FNTransform into the path expressions of PL4XML, a grammar given in a file `File_S` can be applied to an FN item using the following call, which denotes the application as a location step with the axis `transform` and the node test `File_S`:

```
Item_2 := Item_1/transform::File_S
```

6.2 FNTransform on Graph Notation

Field Notation grammars can also be applied to XML elements in graph notation. Both forward and backward axes can be used for selecting and updating sub–items.

In rule heads of grammar rules for field notation it is convenient to use the field notation for splitting an FN–triple into its components. The rule heads of grammar rules for graph notation should, however, contain variables rather than FN–triples.

Comparison of Siblings. In the following, extended XML document we want to mark all `entry` elements by the percentage of change in the `value` attribute compared to the preceding `entry` element (of the same chart):

```
<stocks index="dax100">
   <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
   <chart wkn="600800">
      <entry date="14.12.2002" value="40"/>
      <entry date="15.12.2002" value="50"/>
      <entry date="16.12.2002" value="60"/>
      <entry date="17.12.2002" value="30"/>
   </chart>
</stocks>
```

This can be done very elegantly using an FNG if the document is stored in graph notation in the internal PROLOG database. Then, the preceding `entry` element can be found using the backward axis `preceding_sibling`:

```
:- dislog_variable_set(fn_mode, gn).

Entry ---> Entry :-
   entry := Entry/tag::'*',
   B := Entry@value,
   A := Entry/preceding_sibling::'*'@value,
   compute_percent(A, B, C),
   Entry := Entry*[@change:C].
X ---> X.

compute_percent(A, B, C) :-
   atom_number(A, X), atom_number(B, Y),
   Z is 100 * (Y - X)/X,
   ( Z >= 0, concat(['+', Z, '%'], C)
   ; concat(Z, '%', C) ).
```

The second, third, and fourth `entry` element of the second chart are marked by a corresponding `change`–attribute by the assignment expression "`*[@change:C]`":

```
<stocks index="dax100">
   <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
   <chart wkn="600800">
      <entry date="14.12.2002" value="40"/>
      <entry date="15.12.2002" value="50" change="+25%"/>
      <entry date="16.12.2002" value="60" change="+20%"/>
```

```
      <entry date="17.12.2002" value="30" change="-50%"/>
   </chart>
</stocks>
```

The same marking of the `entry` elements could also be done – less elegantly – by traversing the list of sub–elements of each `chart` element.

7 The Library FnUpdate

This very powerful library it is more complicated to use than FnPath, FnSelect and FnTransform. We will describe it mainly by examples. Currently, FnUpdate only works on field notation.

We will show how to apply update operations to the following stock document:

```
<stocks index="dax100">
   <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
   <chart wkn="600800">
      <entry date="14.12.2002" value="40"/>
      <entry date="15.12.2002" value="50"/>
   </chart>
</stocks>
```

We will refer to the field notation for this document by `$Stocks_2`. For a call "`X := $Stocks_2 ...`" we will list the answers `X` in Xml representation. Each answer becomes a separate Xml element preceeded by a number.

Update Steps. We allow for update steps of the following forms:

```
update::Association_Trees
insert::Association_Trees
delete::Association_Trees
```

They can be abbreviated as follows:

- `/update::Association_Trees` becomes $*$ `Association_Trees`,
- `/insert::Association_Trees` becomes `<+> Association_Trees`,
- `/delete::Association_Trees` becomes `<-> Association_Trees`.

7.1 Update

Updates in Arbitray Chart Element. The following query selects a chart and modifies the value in the entry for `14.12.2002` to `60`:

```
?- X := $Stocks_2/chart *
      [/entry::[@date='14.12.2002']@value:'60'].

1: <chart wkn="200400">
      <entry date="14.12.2002" value="60"/>
   </chart>
2: <chart wkn="600800">
      <entry date="14.12.2002" value="60"/>
      <entry date="15.12.2002" value="50"/>
   </chart>
```

The following query selects a chart and modifies the value in all entries with the value `50` to `60`:

```
?- X := $Stocks_2/chart * [/entry::[@value='50']@value:'60'].

1: <chart wkn="200400">
      <entry date="14.12.2002" value="30"/>
   </chart>
2: <chart wkn="600800">
      <entry date="14.12.2002" value="40"/>
      <entry date="15.12.2002" value="60"/>
   </chart>
```

Update in Selected Chart Element. The following query selects the chart with the wkn `600800` and modifies the value in all its entries to `60`:

```
?- X := $Stocks_2/chart::[@wkn='600800'] *
      [/entry@value:'60'].

1: <chart wkn="600800">
      <entry date="14.12.2002" value="60"/>
      <entry date="15.12.2002" value="60"/>
   </chart>
```

Update with Embedded Computation. The following query selects a chart and increases the value in all its entries by `1`. This is done by selecting the attribute value `V` and by then adding `1` by `add_to_atom` to obtain the resulting value `W`:

```
?- X := $Stocks_2/chart * [
```

```
      /entry::[@value=V, add_to_atom(V, '1', W)]@value:W].

1: <chart wkn="200400">
      <entry date="14.12.2002" value="31"/>
   </chart>
2: <chart wkn="600800">
      <entry date="14.12.2002" value="41"/>
      <entry date="15.12.2002" value="51"/>
   </chart>
```

Updates in all Chart Elements. The following query modifies the values in all entries of all `chart` elements of `14.12.2002` to `60`:

```
?- X := $Stocks_2 *
      [/chart/entry::[@date='14.12.2002']@value:'60'].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" value="60"/>
      </chart>
      <chart wkn="600800">
         <entry date="14.12.2002" value="60"/>
         <entry date="15.12.2002" value="50"/>
      </chart>
   </stocks>
```

The following query adds the attribute assignment `time="10:00"` to all entries of all `chart` elements of `14.12.2002`:

```
?- X := $Stocks_2 *
      [/chart/entry::[@date='14.12.2002']@time:'10:00'].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" time="10:00" value="30"/>
      </chart>
      <chart wkn="600800">
         <entry date="14.12.2002" time="10:00" value="40"/>
         <entry date="15.12.2002" value="50"/>
      </chart>
   </stocks>
```

Adding Sub–Elements. The following query adds a sub–element of the form `<time>10:00</time>` to all entries of all `chart` elements of `14.12.2002`:

```
?- X := $Stocks_2 *
```

```
      [/chart/entry::[@date='14.12.2002']/time:['10:00']].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" value="30">
            <time>10:00</time>
         </entry>
      </chart>
      <chart wkn="600800">
         <entry date="14.12.2002" value="40">
            <time>10:00</time>
         </entry>
         <entry date="15.12.2002" value="50"/>
      </chart>
   </stocks>
```

The following query adds a sub–element with the tag `<time_with_offset>` to all entries of all `chart` elements of `14.12.2002`:

```
?- X := $Stocks_2 * [/chart/entry::[@date='14.12.2002']
      /time_with_offset:[time:[offset:'+01:00']:['10:00']]].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" value="30">
            <time_with_offset>
               <time offset="+01:00">10:00</time>
            </time_with_offset>
         </entry>
      </chart>
      <chart wkn="600800">
         <entry date="14.12.2002" value="40">
            <time_with_offset>
               <time offset="+01:00">10:00</time>
            </time_with_offset>
         </entry>
         <entry date="15.12.2002" value="50"/>
      </chart>
   </stocks>
```

The following query adds a sub–element `<time><hour>10:00</hour></time>` to all entries of all `chart` elements:

```
?- X := $Stocks_2 * [/chart/entry/time/hour:['10:00']].

1: <stocks index="dax100">
      <chart wkn="200400">
```

```
      <entry date="14.12.2002" value="30">
         <time><hour>10:00</hour></time>
      </entry>
   </chart>
   <chart wkn="600800">
      <entry date="14.12.2002" value="40">
         <time><hour>10:00</hour></time>
      </entry>
      <entry date="15.12.2002" value="50">
         <time><hour>10:00</hour></time>
      </entry>
   </chart>
</stocks>
```

7.2 Insertion

The following query adds a new sub–element `<entry date="16.12.2002" .../>` to the `chart` element with the wkn 200400:

```
?- X := $Stocks_2 <+> [/chart::[@wkn='200400']
      /entry:[date:'16.12.2002', value:'60']:[]].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" value="30"/>
         <entry date="16.12.2002" value="60"/>
      </chart>
      ...
   </stocks>
```

In comparison, the following query adds the content `[date:16.12.2002, ...]:[]` to the `entry` element of the `chart` element with the wkn 200400:

```
?- X := $Stocks_2 * [/chart::[@wkn='200400']
      /entry:[date:'16.12.2002', value:'60']:[]].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" value="30">
            [date:16.12.2002, value:60]:[]
         </entry>
      </chart>
      ...
   </stocks>
```

7.3 Deletion

The following query deletes the assignment for the attribute `value` from the `entry` element of `14.12.2002` of the `chart` element with the wkn `200400`:

```
?- X := $Stocks_2 <-> [/chart::[@wkn='600800']
      /entry::[@date='14.12.2002']@value] ].

1: <stocks index="dax100">
      <chart wkn="200400">
         <entry date="14.12.2002" value="30"/>
      </chart>
      <chart wkn="600800">
         <entry date="14.12.2002"/>
         <entry date="15.12.2002" value="50"/>
      </chart>
   </stocks>
```

8 Case Studies

PL4XML has been used in several projects dealing with HTML or XML documents.

Two major applications of FNTransform to Stock Data and to the Jean Paul letters will be shortly sketched in this section. More detailed descriptions can be found in [15] (Stock Data) and [18] (Jean Paul), respectively. Section 8.1 shows how PL4XML can be used for *extracting* stock *data* from Web pages; the relevant stock information is located automatically in the document, such that the extraction process becomes robust to certain changes of the document structure over time. Section 8.2 describes the *transformation* and *annotation* of *linguistic* content in the Jean Paul project.

Further linguistic data processing is currently done in the Adelung project, which is mainly based on Definite Clause Grammars (DCGs) in addition to FNGs. Moreover, we have applied FNQuery for the following purposes:

- for analysing and visualizing rule–based knowledge [17] from the *diagnostic expert system* D3 [12],
- for reasoning about and for refactoring PROLOG and DATALOG rules [16],
- for managing complex structured biological knowledge about signalling pathways, and
- for handling mathematical knowledge represented in the XML languages MATHML and OPENMATH [9].

The case studies about rule–based knowledge, such as diagnostic rules or PROLOG rules, are mainly based on FNQuery, i.e., there are no transformations

or updates. Also the case study for handling complex structured mathematical knowledge is only based on FnQuery. The biological case study about signalling pathways is heavily based on FnQuery and FnUpdate.

8.1 Extracting Stock Data from Web Pages

The *structure of Web sites* can change over time in a highly dynamic way, even though their semantic contents stays the same. For example, a restructuring of an Html page can insert further levels of nesting, such that certain information will be found at a deeper level after the restructuring, or the names of certain elements can be changed.

We have investiagted the Web site of a German company providing stock information: `http://www.finanztreff.de`, and the Web site of the well–known company Amazon: `http://www.amazon.de`. In both cases the goal was to extract some information based on partial (i.e., incomplete) knowledge about the structure of the Html pages.

Extraction of the Stock Table. The following rules can be used for extracting stock data from an Html page. The relevant part of the Html document with the stock data as well as its representation in field notation are given in the appendix. The overall structure of the document is unknown, and it could also vary over time. By looking at the Html page in the browser it can be seen that the page contains a table, such that each row provides stock information about a stock share, and that one of the rows contains the string "Boerse / WKN". An example of such a table is shown in Appendix A and in Figure 3.

The following rules find out where the string "Boerse / WKN" is located in the document `O`, and they return the corresponding path and the table:

```
html_to_path_and_relevant_table(Html, Path, Table) :-
   Table := Html/P1/T1,
   html_table_tag(T1),
   Xs := Table/P2/content::'*',
   member('Boerse / WKN', Xs),
   \+ ( html_table_tag(T2), fn_path_contains(P2, T2) ),
   Path = P1/T1/P2.

html_table_tag(Tag) :-
   member(Tag, ['TABLE', 'Table', table]).
```

This method is very robust, since it will always find relevant paths under the weak assumptions that we have made. Currently, the string "Boerse / WKN" is found under the following path:

```
html/body/p/p/p/'Table'/'TR'/'TD'/table/'TBODY'/'TR'/'TH'
```

Observe, that the tag `'TBODY'` – as well as some other tags – was not present in the original document; it was introduced by the SGML parser of SWI–PROLOG. Surprisingly, the table that we are interested in is located within another table; thus, it would not be enough to just look for a table in the document.

Bezeichnung Börse / WKN	Letzter Umsatz	+/- %	GUmsatz Trend	Zeit Datum	Bid Ask	Vortag Erster	Hoch Tief	Kassa Volumen
Adidas-Salomon XETRA / 500340	80,80 200	-1,27 -1,55	138.951 -=+--+++	18:06:42 06.05.2002	80,61 80,85	82,07 81,75	82,34 80,47	0,00 11.227.241
Allianz XETRA / 840400	253,01 200	0,48 0,19	378.227 ++++-+--	18:24:43 06.05.2002	253,02 253,54	252,53 252,79	254,85 250,85	0,00 95.695.213
BASF XETRA / 515100	45,45 5.500	-0,69 -1,50	1.044.866 =+--=+==	18:12:25 06.05.2002	45,41 45,50	46,14 46,00	46,29 45,38	0,00 47.489.160
Bayer XETRA / 575200	35,52 200	-0,35 -0,98	1.528.250 ---+---+	18:36:36 06.05.2002	35,43 35,53	35,87 35,96	36,28 35,42	0,00 54.283.440
BMW XETRA / 519000	44,61 700	0,36 0,81	1.010.323 +++==+--	18:36:22 06.05.2002	44,57 44,62	44,25 44,29	44,82 44,20	0,00 45.070.509
Commerzbank XETRA / 803200	19,47 500	-0,12 -0,61	1.034.195 --+====-	18:17:47 06.05.2002	19,47 19,52	19,59 19,65	19,84 19,27	0,00 20.135.777
DaimlerChrysler XETRA / 710000	51,50 500	0,29 0,57	2.574.247 +-=--=+-	18:35:02 06.05.2002	51,46 51,50	51,21 51,00	51,60 50,60	0,00 132.573.720
DAX INDEX XETRA / 846900	4.889,98 0	7,21 0,15	0 =+====+=	18:52:31 06.05.2002	0,00 0,00	4.882,77 4.884,03	4.926,78 4.876,58	0,00 0
Degussa XETRA / 542190	34,60 1.200	0,64 1,88	198.350 -+=-==+	18:11:34 06.05.2002	34,54 34,60	33,96 34,40	34,73 34,15	0,00 6.862.910

Figure 3: Stock Data in a WWW Browser

Pruning of the Stock Table. For further processing the stock table we use the file `stock.fng` with the following FNG:

```
'TABLE':_:['TBODY':Es] ---> table:Es.
'TH':_:X ---> th:X.
'TR':_:Es ---> tr:Es.
'TD':_:X ---> td:X.
img:_:X ---> X.
'A':_:X ---> X.
'FONT':_:[X] ---> X.
'BR':_:_ ---> br:[].
X ---> X.
```

The substitution rules were found by iteratively simplifying the structure of the document. As long as the current result contained redundant elements or attributes, these were deleted. Finally, complex elements such as "`'FONT':_:[X]`", "`td:['A':_:X]`", and "`'TD':_:X`" were substituted by simpler ones, namely "`X`" and "`td:X`", respectively. The obtained results are shown in Appendix C.

Originally, we had started by deleting redundant attributes. But after deleting enough redundant attributes a condensed result was obtained, which made it possible to come up with the FNG above.

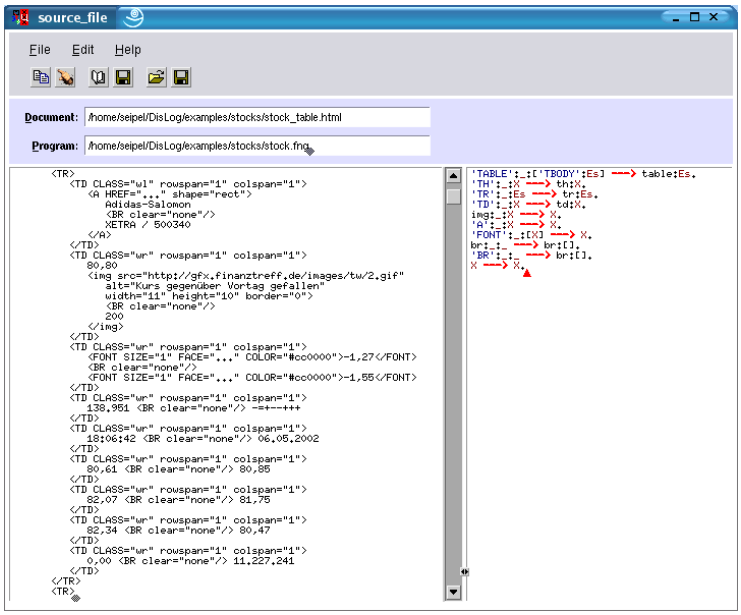

Figure 4: Graphical User Interface of FnQuery

8.2 Jean Paul

We have annotated and transformed letters from and to the well–known German writer *Jean Paul*. We worked with Html files that had been obtained from the original Microsoft Word documents. In the Web browser they looked like follows:

1. Von Erhard Friedrich Vogel. Rehau, 6. Mai 1781, Sonntag
Überlieferung
H: BL, Eg. 2008. 1 Bl. 2°, 1/2 S.
...
Erläuterungen
Erhard Friedrich Vogel, am 17. November 1750 als ältester Sohn
des Bayreuther Hofkammerrates Johann Achatius Vogel ...
...
1, 18-19 **Gehen** bis **sind]** Dem folgenden Brief Vogels ist zu entnehmen, ...

Figure 5: A Letter to Jean Paul

The Html file looks as follows, where we have made some suitable abbreviations to make the structure more visible: By `<font ...>` we abbreviate `<font face="Times New Roman" size="3">`, and by `<p ...>` we abbreviate `<p style="margin-top:0;margin-bottom:0;">`.

```
<html> ...
<body>
<p style="margin-bottom:0;">
   <font ...><em>1. Von Erhard Friedrich Vogel.
      Rehau, 6. Mai 1781, Sonntag </em></font></p> ...
<p ...><font ...>Ueberlieferung</font></p>
<p ...><font ...>H: BL, Eg. 2008. 1 Bl. 2, 1/2 S. </font></p> ...
<p ...><font ...>Erlaeuterungen</font></p>
<p ...><font ...>Erhard Friedrich Vogel,
   am 17. November 1750 als aeltester Sohn des Bayreuther
   Hofkammerrates Johann Achatius Vogel ... </font></p> ...
<p ...><font ...><em>1, </em>18-19
   <strong>Gehen </strong> bis <strong>sind] </strong>
   Dem folgenden Brief Vogels ist zu entnehmen, ... </font></p> ...
</body>
</html>
```

These HTML documents have been transformed to XML documents using a micture of FNGs and Definite Clause Grammars (DCGs). A fragment of the used FNG is shown in the following:

```
body:_:Es ---> comment:Es.
p:_:Es ---> notep:Es.
font:_:[X] ---> X.
em:_:Es ---> commentHead:Es :-
   retract(letterHeadFlag).
em:_:Es ---> page:Es.
strong:_:Es ---> lemma:Es.
```

Observe that the first rule for `em` only fires, if there is a fact `letterHeadFlag` in the PROLOG database, which indicates that we are processing the head of the letter. Subsequent `em` elements are transformed to `page` elements, since the fact has been retracted.

```
<comment>
   <notep>
      <commentHead>1. Von Erhard Friedrich Vogel.
         Rehau, 6. Mai 1781, Sonntag
      </commentHead> </notep>
   <ednote type="Ueberlieferung">
      <notep>H: BL, Eg. 2008. 1 Bl. 2, 1/2 S.</notep> ... </ednote>
   <ednote type="Erlaeuterungen"> ...
      <notep>
         <page>1,</page> 18-19
         <lemma>Gehen</lemma> bis <lemma>sind]</lemma>
         Dem folgenden Brief Vogels ist zu entnehmen, ...
      </notep> </ednote>
</comment>
```

The structuring into `<ednote type="Type">` for `Type = Ueberlieferung"` and `Type = "Erlaeuterungen"` can be done using DCGs. Also a `remark` element is formed using DCGs.

```
<note id="23">
   <remark>
      <position page="1" line="18-19"/>
      <lemma>Gehen</lemma> <lemma>sind</lemma>
   </remark>
   Dem folgenden Brief Vogels ist zu entnehmen, ...
</note>
```

Finally, a presentation layer with *navigation utilities* was created using FnUpdate.

9 Conclusions and Future Work

Pl4Xml shows that techniques for processing and reasoning about complex Xml documents can be integrated nicely into Prolog applications. Thus, it becomes possible to formulate complex queries to Xml documents in a compact and intuitive way. Instead of embedding an Xml query language into a procedural language, we propose a homogeneous framework for accessing, transforming and reasoning about Xml documents in a declarative environment.

The first library FnQuery had been developed for Xml documents in field–notation and later extended to graph notation. Currently, we are planning to investigate optimzation techniques for the evaluation of queries and mixed data representations that can alternate between field and graph notation.

Currently, the predicates of FnQuery are evaluated in a top–down and tuple–oriented fashion using Prolog; for computing all answers to a query we use backtracking. For more complicated rule systems containing recursive rules we are planning to use set–oriented bottom–up evaluation as in Datalog.

References

[1] *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and Xml, Morgan Kaufmann, 2000.

[2] *F. Bancilhon, S. Cluet, C. Delobel:* Query Languages for Object–Oriented Database Systems: the O_2 Proposal, Proc. 2nd Intl. Workshop on Database Programming Languages, 1989.

[3] *J. Baumeister, D. Seipel, F. Puppe:* Refactoring Methods for Knowledge Bases, Proc. 14th International Conference on Engineering Knowledge in the Age of the Semantic Web EKAW 2004, Springer, LNAI 3257, pp. 157-171.

[4] *J. Baumeister, D. Seipel:* Smelly Owls – Design Anomalies in Ontologies, Proc. 18th International Florida Artificial Intelligence Research Society Conference FLAIRS 2005, AAAI Press, 2005, pp. 215–220.

[5] *T. Berners–Lee, J. Hendler, O. Lassila:* The Semantic Web, Scientific American, May 2001.

[6] *F. Bry, S. Schaffert:* Pattern Queries for XML and Semistructured Data. Proc. 17th Workshop Logische Programmierung, 2002.

[7] *S. Ceri, G. Gottlob, L. Tanca:* Logic Programming and Databases, Springer, 1990.

[8] *M. Gross–Hardt:* Querying Concepts — An Approach to Retrieve XML Data by Means of their Data Types. Proc. 17th Workshop Logische Programmierung, 2002.

[9] *B. Heumesser, D. Seipel, U. Güntzer:* Flexible Processing of XML Based Mathematical Knowledge in a PROLOG Environment, Proc. Intl. Conf. on Mathematical Knowledge Management MKM'2003, Springer LNCS (to appear).

[10] *M. Hopfner, D. Seipel, J. Wolff von Gudenberg:* Comprehending and Visualising Software based on XML–Representations and Call Graphs, Proc. 11th IEEE International Workshop on Program Comprehension IWPC 2003.

[11] *M. Kifer, G. Lausen:* F–Logic: A Higher–Order Language for Reasoning about Objects, Proc. ACM SIGMOD Conference on Management of Data, 1989.

[12] *P. Puppe et al.:* D3. `http://d3web.informatik.uni-wuerzburg.de/`

[13] *D. Seipel:* DISLOG – A Disjunctive Deductive Database Prototype, Proc. 12th Workshop on Logic Programming WLP 1997.

[14] *D. Seipel:* The DISLOG Developers' Kit (DDK), `http://www1.informatik.uni-wuerzburg.de/databases/DisLog`

[15] *D. Seipel:* Processing XML Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.

[16] *D. Seipel, M. Hopfner, B. Heumesser:* Analyzing and Visualizing Prolog Programs based on XML Representations. Proc. International Workshop on Logic Programming Environments WLPE 2003.

[17] *D. Seipel, J. Baumeister, M. Hopfner:* Declarative Querying and Visualizing Knowledge Bases in XML, Proc. 15th International Conference on Applications of Declarative Programming and Knowledge Management INAP 2004, pp. 140-151.

[18] *D. Seipel, K. Prätor:* XML Transformations Based on Logic Programming. Proc. 18th Workshop on Logic Programming WLP 2005, pp. 5–16.

[19] *V. Vianu:* A Web Odyssey: from Codd to XML, Proc. Intl. Conference on Principles of Database Systems PODS'2001.

[20] *J. Wielemaker:* SWI–PROLOG 5.0 Reference Manual, `http://www.swi-prolog.org/`

[21] *J. Wielemaker, A. Anjewierden:* Programming in XPCE/PROLOG, `http://www.swi-prolog.org/`

[22] *J. Wielemaker:* SWI–PROLOG SGML/XML Parser, Library Manual, 2002.

[23] XML Query Requirements, Working Draft of the World Wide Web Consortium (W3C), `http://www.w3.org/TR/2001/WD-xmlquery-req-20010215`.

Appendix A. Stock Data in HTML

The following HTML–table occurs deeply nested within the HTML–document

```
http://www.finanztreff.de/portal/
   kurse_index_listen.htm?u=0&p=0&k=0&symbol=DAX
```

```
<TABLE BORDER=0 CELLPADDING=2 cellspacing=2
       WIDTH=693 BGCOLOR="#c6c6c6">
   <TR>
      <TH CLASS="hl">Bezeichnung<BR>Boerse / WKN</TH>
      <TH CLASS="hl">Letzter<BR>Umsatz</TH>
      <TH CLASS="hl">+/-<BR>%</TH>
      <TH CLASS="hl">GUmsatz<BR>Trend</TH>
      <TH CLASS="hl">Zeit<BR>Datum</TH>
      <TH CLASS="hl">Bid<BR>Ask</TH>
      <TH CLASS="hl">Vortag<BR>Erster</TH>
      <TH CLASS="hl">Hoch<BR>Tief</TH>
      <TH CLASS="hl">Kassa<BR>Volumen</TH>
   </TR>
   <TR>
      <TD CLASS="wl">
         <A HREF="javascript:var ...">Adidas-Salomon</A> <BR>
         XETRA / 500340
      </TD>
      <TD CLASS="wr">
         82,07
         <img src="http://gfx.finanztreff.de/images/tw/1.gif"
              alt="Kurs gegenber Vortag gestiegen"
              width=11 height=10 border=0> <BR>
         0
      </TD>
      <TD CLASS="wr">
         <FONT SIZE="1" FACE="Verdana, Arial, Helvetica"
               COLOR="#009900"> 1,57 </FONT> <BR>
         <FONT SIZE="1" FACE="Verdana, Arial, Helvetica"
               COLOR="#009900"> 1,95 </FONT>
      </TD>
      <TD CLASS="wr">550.706<BR>=-+=====</TD>
      <TD CLASS="wr">21:15:17<BR>03.05.2002</TD>
      <TD CLASS="wr">0,00<BR>0,00</TD>
      <TD CLASS="wr">80,50<BR>80,00</TD>
      <TD CLASS="wr">82,90<BR>80,00</TD>
      <TD CLASS="wr">0,00<BR>45.196.441</FONT></TD>
   </TR>
   <TR> ... </TR>
   ...
</TABLE>
```

Appendix B. Pruned Stock Data

We have obtained the following pruned table. The first row contains the attributes from the header of the original HTML–table, and each subsequent row contains the corresponding data for one stock share.

```
<table>
   <tr>
      <th> Bezeichnung <br/> Boerse </th>
      <th> Letzter <br/> Umsatz </th>
      <th> +/- <br/> % </th>
      <th> GUmsatz <br/> Trend </th>
      <th> Zeit <br/> Datum </th>
      <th> Bid <br/> Ask </th>
      <th> Vortag <br/> Erster </th>
      <th> Hoch <br/> Tief </th>
      <th> Kassa <br/> Volumen </th>
   </tr>
   <tr>
      <td> Adidas-Salomon <br/> XETRA / 500340 </td>
      <td> 82,07 <br/> 0 </td>
      <td> 1,57 <br/> 1,95 </td>
      <td> 550.706 <br/> =-+===== </td>
      <td> 21:15:17 <br/> 03.05.2002 </td>
      <td> 0,00 <br/> 0,00 </td>
      <td> 80,50 <br/> 80,00 </td>
      <td> 82,90 <br/> 80,00 </td>
      <td> 0,00 <br/> 45.196.441 </td>
   </tr>
   ...
</table>
```

By accumulating the data of several days we can obtain the following stock charts:

```
<stocks index="dax100">
   <chart wkn="500340" name="Adidas-Salomon">
      <entry date="03.05.2002" value="82,07"/>
      <entry date="06.05.2002" value="80,80"/>
      <entry date="08.05.2002" value="84,85"/>
   </chart>
   <chart wkn="519000" name="BMW">
      ...
   </chart>
   ...
</stocks>
```