statistical data analysis, linear optimization and dynamic programming are used for dimension reduction. But in same cases it is possible to lead the task to linear or quadratic optimization. Solution and solution method substantiation of given task are planed courses of author investigation.

Bibliography

[1] G. Dantzig, *Linear Programming and Extensions.* Princeton University Press, Princeton, 1963.

[2] G. Bruno, and A. Gilio, *Application of the simplex method to the fundamental theorem for the probabilities in the subjective theory.* Bologna, Statistica, 40(3), 1980. pp. 337–344.

Transformation of XML Documents with Prolog

René Haberland and Igor L. Bratchikov

Saint Petersburg State University, Faculty of Applied Mathematics and Control Processes

Abstract

Transforming XML documents with conventional XML languages, like XSL-T, is disadvantageous because there is too lax abstraction on the target language and it is rather difficult to recognize rule-oriented transformations. Prolog as a programming language of declarative paradigm is especially good for implementation of analysis of formal languages. Prolog seems also to be good for term manipulation, complex schema-transformation and text retrieval.

In this report an appropriate model for XML documents is proposed, the basic transformation language for Prolog LTL is defined and the expressiveness power compared with XSL-T is demonstrated, the implementations used throughout are multi paradigmatic.

Contents

1   Introduction

Transforming XML documents means to build a target XML document from a source XML document by rebuilding former parts and including new parts by querying external data. Declarative approaches on transforming XML documents already exist in the functional paradigm. XSL-T and XQuery represent those XML transforming languages that hold semantic equivalence by avoiding side effects. Alternatively imperative approaches like [6] have been proposed. Both approaches lack either from abstraction of the transformed language or from side effects. Abstraction means that there ought to be a distinction between XML as output and transformation descriptions that operate on it. The ideal would be to obtain transform rules "when *Pattern* then *Substitute*". Those disadvanteges hinder the validation of composed transformation rules because the rules become unreadable and complex. Side effects could cause assertion violations and even so the orthogonality of the transformation can not be guaranteed and finally the target document becomes falsely generated.

Nevertheless a logical approach must injure referential transparency due unifying terms. Side effects and predicates constrain semantics to be denoted not in a functional but in a predicative style. A logical approach will also be of interest because no proper transformation language has been proposed yet. The non linear character predicates can be evaluated seems to be promising in reforming more compact transformations. Therefore it can be assumed a shorter and more abstract transformation is obtainable. The more abstract a transformation becomes the easier it gets to specify it. It can also be stated that using a logical transformation approach simplifies the acquisition because queries to XML documents already are transformations. This way it is possible to avoid semantical anomalies

such as in XSL-T that depends on XPath as its query language for external XML sources but itself is not describable in XML. The result of previously determined results is implicitly propagated to its descending child nodes. In conclusion the following threads are going to be researched:

- Find a base of transform and query operators.

- Simplify transformations that need many queries to build similiar blocks into the target document.

- Investigate cases in which inverse predicates serve as validator.

- Push up limits from recent XPath and XSL-T reference implementations concerning syntactical constraints, e.g. user defined functions, closed world and multi indices [10, 11].

The aim of this article is to define a logical language for XML processing based on Prolog and to evaluate this logical transformation language (LTL) in comparison to the clear XML transformation language XSL-T. Primarily the comparison is oriented to user requirements formulated in today's XSL-T forums. It is intended LTL is extensible as much as possible to break through the limited syntax of XSL-T. The task restricts to XML documents, but essentially all hierarchical markup documents are affected. Domains of a LTL are complex schema-transformations, text retrieval and the analysis of formal languages.

First work on logical transformations on XML documents has partially been done in [1]. Unfortunately the described mismatch between query and target languages proceeds and most of defined predicates just reimplement a functional framework attempt. Consequently predicates became tough combinators and the evaluation order got inflexible because profound language features of Prolog missed. Therefore combinators became complex, numerous, redundant and finally error-prone.

2   Importance

The way knowledge is represented and infered logical is one common approach. Considering Prolog as logical programming language transformation rules can directly be represented as Horn clauses. In distinction to

functional programming languages [3–5, 9] a premise might be partially undetermined. In fact a premise predicate might contain symbols unified after depending conjunctions have been evaluated or even never. Prolog makes use of side effects to unify terms. Although these anomalies are locally to each Horn clause which makes Prolog attractive for componing. Summarizing Prolog knowledge can be represented by facts and Horn clauses. This is essential comparing Prolog with XQuery which allows exclusively rigid FLOWR expressions. Shorter code affects a better formalisation of the transforming semantics and allows simpler checkings against given specifications.

It is notable that APIs have been developed [8] that enable bindings to but also from Java. Therefore numerous feasibilities result an integration of previous XML processors. The opposite direction permits sophisticated algorithms in Java been rewritten in Prolog. That might be capable for descriptive transformation languages, business rules policies, multi channel publishing etc.

Starting with an appropriate data model for LTL, a rather simple derivation rules will be defined later on.

3   Data Model

Well-formed XML documents at least consist of element, text and attribute nodes (figure   1). It can be assumed that parsed and serialized documents are well-formed and canonized hence valid normalforms simplify analysis of duplicates. Canonized documents describe documents which contain attributes that have been sorted in lexicographic ascending order by attribute name. Standard Prolog has no static type checking, so syntactical invalid transformations unfortunately can not be detected before serializing the target document. Since Prolog does not offer pointer references it gets necessary to specify nodes completely as short as possible in its full specification. Lists seem fairly to represent attribute and children sets. Only hedges or children sets must satisfy a strict order. Analogous defined nodes could be rewritten in lists as it was done by [1]. But this attempt lack on proving node's type and consequently does not bring any advantage. Despite an uniform syntax affected predicates would become even more complex because precise distinction between list members would be necessary.

$\langle\text{hedge}\rangle \rightarrow \varepsilon | \langle\text{node}\rangle | \langle\text{node}\rangle, \langle\text{hedge}\rangle$
$\langle\text{node}\rangle \rightarrow \langle\text{element}\rangle | \langle\text{textnode}\rangle | \langle\text{pi}\rangle | \langle\text{comment}\rangle$
$\langle\text{element}\rangle \rightarrow \textbf{element}(\langle\text{name}\rangle, [\langle\text{attributes}\rangle], [\langle\text{hedge}\rangle])$
$\langle\text{textnode}\rangle \rightarrow \textbf{text}(\ \langle\text{text}\rangle\ )$
$\langle\text{pi}\rangle \rightarrow \textbf{pi}(\ \langle\text{text}\rangle\ )$
$\langle\text{comment}\rangle \rightarrow \textbf{comment}(\ \langle\text{text}\rangle\ )$
$\langle\text{name}\rangle \rightarrow \langle\text{alpha}\rangle \langle\text{name2}\rangle$
$\langle\text{text}\rangle \rightarrow \langle\text{name2}\rangle$
$\langle\text{name2}\rangle \rightarrow \varepsilon | \langle\text{alphanum}\rangle \langle\text{name2}\rangle$
$\langle\text{alphanum}\rangle \rightarrow \langle\text{alpha}\rangle | \textbf{0}|\textbf{1}|\textbf{2}|\textbf{3}|\textbf{4}|\textbf{5}|\textbf{6}|\textbf{7}|\textbf{8}|\textbf{9}$
$\langle\text{alpha}\rangle \rightarrow \textbf{a}|..|\textbf{z}$
$\langle\text{attributes}\rangle \rightarrow \varepsilon | \langle\text{attribute}\rangle | \langle\text{attribute}\rangle, \langle\text{attributes}\rangle$
$\langle\text{attribute}\rangle \rightarrow \textbf{'}\langle\text{name}\rangle =''\langle\text{text}\rangle''\textbf{'}$

Figure 1: *Backus-Naur form of Prolog data model representing XML documents. The superior node in a valid XML document has to be an element node.*

In conclusion we get a more intuitive and compact representation that avoids mandatory writing of new combinators even for simple transformations. Furthermore there is no dependence on certain transformation orders due control can be influenced by individual predicates.

To describe any XML document it is enough to use element and text nodes.

This statement first induces that all other nodes - processing instruction and comment - can be expressed just by unambiguous element and text nodes (cmp. fig. 1).

To show this it is sufficient to find some unambiguos encodings σ, τ and to expand the alphabet by some special symbols $\{\pi, \kappa\}$ with $\sigma : pi(\underline{text}) \mapsto \pi \cdot \underline{text}$ and $\tau : comment(\underline{text}) \mapsto \kappa \cdot \underline{text}$. The binary operation $\cdot$ concates two strings.

Further it states that attributes in element nodes can also be rewritten. Hence attribute nodes contain two arguments it is obvious to use a tagged encoding such as element nodes. So an appropriate encoding would be: $\alpha : \ '\underline{name} =''\underline{text}''\ ' \mapsto element(\underline{name}, [], [text(\omega \cdot \underline{text})])$. The element node of the right side specifies the upper element node, so it is necessary to agree upon inserting first all attributes.

Such applying σ, π, α to a given XML document a XML document

results that consists only of element and text nodes. This function obviously is injective. New node types are encoded in analogy. Nodes from different name spaces are handled as usual element nodes with a prefixed name $\Box$.

Given a XML document X, $\forall u, v \in X : reachable(u, v)$ is satisfied.

As it was shown earlier it is sufficient to consider the simplified case. Further such a top level element by induction represents a n-way tree. From the tree character of X' the lenght of any path in X' is bounded by the number of nodes decreased by one $\Box$.

Combinators defined on XML documents can be eliminated in Prolog.

As seen in the previous statement the path can be build just by ascending and descending relationships considering a bidirectional tree. The direction depends on the unification of unevaluated node constructors in Prolog. So, combinators are sufficient but are not necessary $\Box$.

Prolog gives universal operators for transformations for free. Additional operators are not necessary from the computational view.

When Prolog's rules are interpreted as tables a relational algebra equivalent to Codd's algebra can be defined as following. R, S and T describe any relations.

$$\begin{array}{rl}
\textbf{Union } T = R \cup S: & t(x_1, ..., x_m) : -r(x_1, ..., x_m). \\
& t(y_1, ..., y_n) : -s(y_1, ..., y_n). \\
\textbf{Difference } T = R/S: & t(x_1, ..., x_n) : -r(x_1, ..., x_n), not(s(x_1, ..., x_n)). \\
\textbf{Carthesian } T = R \times S: & t(x_1, ..., x_m, y_1, ..., y_n) : -r(x_1, ..., x_m), \\
& s(y_1, ..., y_n). \\
\textbf{Projection } T = \Pi_S(R): & t(s_1, ..., s_n) : -r(x_1, ..., x_m). \\
& \forall s_i \in \{x_1, ..., x_m\} \\
\textbf{Selection } T = \sigma_S(R): & t(x_1, ..., x_n) : -r(x_1, ..., x_n), s(x_1, ..., x_n). \\
\textbf{Renaming } T = \rho_S(R): & t(x_1, ..., x_n) : -r(x_1, ..., x_n).\Box
\end{array}$$

Deletion, Insertion and Exchange of elements in a XML documents requires one step in Prolog.

These three operations correlate with a similar resulting document within a binary relation Ψ. Among the arguments in the document unification in Ψ matches exactly except the changed element, so the common unificator contains exactly one substitution hence hedge manipulation can be fulfilled in one step in terms in Prolog $\Box$.

Since it is not necessary to fully specify the part that has not been changed in Prolog, the number of manipulations is bound by a constant.

There is no need to reconstruct invariant parts like in XSL-T. Such manipulating operations significantly improve the performance of the transformation and are called *non-monotonic* due they avoid permanent input and output transfers.

4 Transformation Rules

The specification of the most important transform operators can be found in the appendix. Depending on their arity the operators can be chained to a path expression analogous to XPath. In distinction to XSL-T the object and meta languages match with each other. Nodes are constructed and queried just in Prolog.

The search strategy of Prolog changes the instantiation process of the target document such that alternatives will only be cut manually. Otherwise combinatorical parts will cause an exponential amount of valid target documents even if the user is only interested in the first element of each result set. For instance, it can be assumed that the LTL would be exactly implemented as defined by inference rules in the appendix. Hence there is no need to stop derivation after first matching of the third ($//$) rule, the search for alternatives could be continued with the fourth ($//$) rule. In such a case the derivation of the template

```
template(element(top,_,[A,A]),[text(T)]):-
    A=element(a,_,_),transform(A//p#1,T).
```

which is semantically similar to the XSL-T stylesheet

```
<xsl:template match="top[count( child::*)=2] and a[1] and a[2]">
    <xsl:text>
        <xsl:value-of select="//a//p"/>
    </xsl:text>
</xsl:template>
```

would return a multiset of text nodes. In comparison, the given XSL-T example would stop in every case even if `value-of` throws an exception. In fact it is not necessary within the implementation to consider alternatives after one matching succeeds, so red cuts were set. Further as it

can be seen in the previous Prolog template it would be rather simple to specify excactly the hedge [A,A,B,A] in top for any element node B.

Delete operators as specified in the appendix tend to shorten the filling of target documents due reducing operations that are not necassary through copying unmodified fragments. Hence in worst case the whole document becomes empty and due to the undecidability of the halt problem, a trade-off concerning the usage of non-monotonic and average operators has to be done by the user. Even in the worst case the problem complexity is restricted by the XSL-T boundary. It should also be taken into account that deleting any element node by a recursive path expression, as it was not done here, affects an ambiguous validation since direct relationships get lost.

5　Comparison

Expressiveness. To compare Prolog with XSL-T it is useful to introduce in XPath equivalent operators. For control structures no special operators were needed because they fit in Prolog's declarative recursion schema.

As announced earlier compact node specification, comfortable aggregations, and the usage of *normal* and non-monotonic predicates enables shortened notations. All except ascending operators navigate within a given root document.

The built-in data structures of Prolog as atoms, tupels, lists and symbols widen XSL-T's default element hedge type. In fact in XSL-T node expressions often have to be converted to desired types. Most conversions have been covered by the path language XPath. Despite this standard Prolog unfortunately has no static type system which significantly diminishes practise acceptance to application programmers. Prolog avoids unnamed predicates and requieres predicates to be called with a complete parameter list. Predicates that were called with less arguments as allowed in XSL-T were not evaluated or were evaluated with a body associated different to the intented body hence scoping of previous definitions is permitted. In general it is rather difficult to define functions by the user for reuse in XSL-T except namespaces and it is even more difficult to find an adequate representation that narrows its mathematical description. So it is quite impossible to formulate an adequate attribute sorting due the restrictiveness and invariance of built-in functions of XPath in XSL-T version 1.0.

Usability. The Well-Formedness of documents is not guaranteed neither by XSL-T nor by Prolog. So XSL-T can generate hedges instead of one top level element node and affect an unsuitable output by choosing an incorrect output encoding. Prolog can also be effected by similar effects, but hedges on top niveau will overwrite previous alternatives successively. To investigate the behavior of Prolog in comparison to XSL-T a measurement of common characteristics was done. Therefore so called *Hålstead metrics* were used [7]. Originally they are applied to procedural programming languages. But LTL and XSL-T also have command tags which can be interpreted as operators η_1 and operands η_2 that are described by tag attributes. In more detail tag attributes could also be parsed but this should be avoided so far since results shall not be fixed to one special query language but rather to a transformation language. Hålstead metrics were primary not designed for markup processing, but hence markup is transformed on a higher level of abstraction in a command-like notation, it is worth to notice quantitative results. Used Hålstead metrics include measured program length N, theoretical program length N_T and intellectual level L and language abstraction niveau λ that all base only on η_1, η_2 and an average estimation for failure made by an average-talent application programmer (see appendix). The experimental base counts about 70 different examples that mostly were taken from various XSL-T tutorials and completed by some individual test cases. Each example had two realizations in LTL, one that used exclusively predicates without templates and one that used templates as far as possible.

In common Prolog schema were about 50% shorter than XSL-T variants according to Lines of Code, λ and $\Delta_N = \|N_T - N\|$. This means Prolog programs are significantly more readable than XSL-T stylesheets. Also $N_T : N$ and $\eta_1 : \eta_2$ showed that Prolog is approximately 30% more functional than XSL-T because their relations are more balanced. It can also be deduced that code becomes more difficult in Prolog because of the higher language abstraction niveau. Explicit traverses were almost always shorter than the variants using templates.

Design. XSL-T lacks of closedness. There is no proper method in version 1.0 to define neither native XPath functions nor adequate notation in XSL-T. Contrary, Prolog permits in-script definitions that are appropriate and modularized among libraries or scoped namespaces.

Escpecially in layered architectures for Prolog-Java APIs Java function-
ality and universality can comfortable be plugged into existing higher
level code.

   Implementation. An implementation has been realised using tuPro-
log [8] and Java for String operations. The transformation is a query to
the existing knowledge base which consists at least of the basic operators
defined in the appendix. For example the user is allowed to write some
transformation rules in Prolog as it was done in section 4. Some template
will be processed as soon as some node in the input XML file matches
with the first parameter `element(top,_,[A,A])` where `A` is the element
node `a` specified by `A=element(a,_,_)`. As result the first occurence of
a descended element node `p` is found and its associated text node is re-
turned as secound argument in the template rule. Beneath nodes of top
will not be traversed if it is not specified explicitly by new `template` calls
in a template. The traversing continues with sibling nodes relatively to
the matched node.

6   Conclusion: Error Detection and Correc-
tion

Error detection, diagnostic and correction are one of the most important
tasks of formal language analysis. For performing Prolog-based transfor-
mations of XML document the user has to write appropriate programs
in Prolog. It can cause errors in the target documents owing to lack of
experience of the user. There are two approaches to diminish the proba-
bility of this nuisance. The first one is to design and implement a special
interface that will be able to save the user from programming necessary
transformations in Prolog. It is desirable that such interface would be
easy for application and not to require knowledge of Prolog. Naturally
some converter should be designed for transformation of texts formed in
interface into appropriate Prolog programs. The second approach is to
analyze the target document in order to detect errors. We suppose the
best methods of detecting errors are global ones as they find minimum of
errors and secure syntactic validity of the document after their correction.
Now it is difficult to prefer one of these approaches. Both require further
investigations.

Bibliography

[1] Dietmar Seipel. Processing XML Documents in PROLOG. Proc. 17th
Workshop on Logic Programming. WLP2002.
http://www-info1.informatik.uni-wuerzburg.de/database/papers/wlp_2002.ps.gz

[2] Wim Janssen, Alexandr Korlyukov, Jan Van den Bussche. On the tree-
transformation power of XSLT. arXiv.org 2006
http://arxiv.org/pdf/cs.PL/0603028

[3] Dimitre Novatchev. The Functional Programming Language XSLT -
A proof through examples. November 2001
http://www.topxml.com/xsl/articles/fp

[4] Oleg Kiselyov, Shriram Krishnamurthi. SXSLT: Manipulation
Language for XML. Proc. Fifth Intl. Sym. Practical Aspects of
Declarative Languages. PADL 2003
*http://www.cs.brown.edu/~sk/Publications/Papers/Published/kk-
sxslt*

[5] Phil Wadler. A formal semantics of patterns in XSLT Markup Tech-
nologies. CiteSeer 1999 *http://citeseer.ist.psu.edu/204315.html*

[6] Xact XML-Transformation Framework for Java.
http://www.brics.dk/Xact

[7] Hålstead, Maurice H. Elements of Software Science, Operating, and
Programming Systems Series Volume 7. New York, NY: Elsevier, 1977.

[8] Enrico Denti, Andrea Omicini, Alessandro Ricci. tuProlog: A Light-
weight Prolog for Internet Applications and Infrastructures - Practi-
cal Aspects of Declarative Languages. 3rd International Symposium
(PADL'01). Springer-Verlag 2001

[9] XQuery 1.0 Specifikation, 2006.
http://www.w3c.org/TR/xquery

[10] XSLT 1.0 Specification, 1999.
http://www.w3.org/TR/xslt

[11] XPath 1.0 Specification, 1999.
http://www.w3.org/TR/xpath

APPENDIX

7   Metrics

$N_T = \eta_1 ld(\eta_1) + \eta_2 ld(\eta_2)$
$\lambda = V * L$
$V = Nld(\eta)$

8   Operators

8.1   Aggregations

$$last_E \frac{E = element(\_,\_,[\cdots,C])}{C} \qquad count_E \frac{E = element(\_,\_,[C_1,...,C_n])}{n}$$

$$level_{X_0,X_n} \frac{\begin{array}{c} X_0 = element(\_,\_,[\cdots,C_{0,i_0},\cdots]), C_{0,i_0} = X_1 \\ X_1 = element(\_,\_,[\cdots,C_{1,i_1},\cdots]), C_{1,i_1} = X_2 \\ \vdots \\ X_{n-1} = element(\_,\_,[\cdots,C_{n-1,i_{n-1}},\cdots]) \\ C_{n-1,i_{n-1}} = X_n = element(\_,\_,\_) \end{array}}{[i_0,i_1,\cdots,i_{n-1}]}$$

8.2   Non monotonic

$$copy_X \frac{X = element(\_,\_,\_)}{X}$$

$$copy\_of_X \frac{X = element(N,A,C)}{element(N,A,[])}$$

$$remEl_{E,N} \frac{\begin{array}{c} E = element(Name,A,C) \\ \texttt{append}(Pre,[element(N,\_,\_)|Post],C) \\ \texttt{append}(Pre,Post,C2) \end{array}}{element(Name,A,C2)}$$

$$rem_{E,N} \frac{\begin{array}{c} E = element(Name,A,C) \\ \texttt{append}(Pre,[N|Post],C) \\ \texttt{append}(Pre,Post,C2) \end{array}}{element(Name,A,C2)}$$

8.3   XPath equivalents

$$/_{E,N} \; \frac{E = element(\_, \_, [\cdots, element(N, A, C), \cdots])}{element(N, A, C)}$$

$$?_E \; \frac{E = pi("X")}{'X'}$$

$$//_{E,N} \; \frac{E = element(N, \_, \_)}{E}$$

$$//_{E,N \neq X} \; \frac{E = element(X, \_, [])}{\text{fail}}$$

$$\#_E \; \frac{E = text("X")}{'X'}$$

$$//_{E,N \neq X} \; \frac{E = element(X, \_, [H|T])}{H//N}$$

$$id_{E,Val} \; \frac{E = element(\_, [], \_)}{\text{fail}}$$

$$//_{E,N \neq X} \; \frac{E = element(X, \_, [H|T])}{element(X, \_, T)//N}$$

$$@_{E,Att} \; \frac{E = element(\_, [], \_)}{\text{fail}}$$

$$@_{E,Att} \; \frac{E = element(\_, [\cdots, 'Att="Val" ', \cdots], \_)}{'Val'}$$

$$id_{E,Val} \; \frac{E = element(\_, [\cdots, 'Att="Val" ', \cdots], \_)}{'Att'}$$

$$descendant_E \; \frac{E = element(\_, \_, [])}{\text{fail}}$$

$$descendant_E \; \frac{E = element(\_, \_, \_) \quad E2 = E/\_}{\text{E2}// \_}$$

$$child_{E,C} \; \frac{E = element(\_, \_, [\cdots, C, \cdots])}{C}$$

Detection of Links between Words in the Task of Syntactic-Semantic Analysis of Russian Texts.

Dmitry V. Merkuryev

Department of Computer Science,
Faculty of Mathematics and Mechanics,
Saint-Petersburg State University

28, Universitetsky prospekt, Peterhof, St. Petersburg, 198504, Russia

E-mail: d.merkuryev@mail.ru

Abstract

Professor V. A. Tuzov's functional theory offers the solution for the problem of natural language formalization. The syntactic-semantic analyzer based on this theory is able to solve word sense disambiguation problem for the most sentences of newspaper and even literature Russian texts. The basic principles of the theory are briefly discussed in the paper.
The detection of links between words is one of the most significant operations of the analyzer. This operation allows getting appropriate semantic alternative of a word in a sentence context. The main aspects of links detection are described in the paper.

Contents