A Rule-Based Approach to XML Processing and Web Reasoning *

Jorge Coelho[1], Besik Dundua[2], Mário Florido[2], and Temur Kutsia[3]

[1] ISEP & LIACC, Porto, Portugal
jcoelho@liacc.up.pt
[2] DCC-FC & LIACC, University of Porto, Portugal
{bdundua,amf}@dcc.fc.up.pt
[3] RISC, Johannes Kepler University, Linz, Austria
kutsia@risc.uni-linz.ac.at

Abstract. We illustrate the potential of conditional hedge transformations in Web-related applications on the example of PρLog: an extension of logic programming with advanced rule-based programming features for hedge transformations, strategies, and regular constraints.

1 Introduction

The rule-based approach has been used extensively in many fields, such as expert systems, theorem proving, tree automata, software building and configuration, banking systems, just to name a few. In recent years, the rule-based approach has been experiencing growing popularity in Web applications. One could mention document processing and Web reasoning as prominent examples. The REWERSE project [23] provides an extensive reference material on those topics.

The goal of this paper is to illustrate the potential of strategy-based conditional hedge transformations in Web-related applications. To achieve this goal, first, we present a practical tool: an extension of logic programming with advanced rule-based programming features for hedge transformations, strategies, and regular constraints. Second, we show how it can be used it XML querying, validation, and some basic Web reasoning.

The tool we describe here is PρLog [13] (pronounced Pē-rō-log). It is a Prolog implementation of the ρLog calculus [19], which extends the host language with strategic conditional transformation rules. These rules (basic strategies) define transformation steps on hedges. (A hedge is a sequence of unranked terms.) Strategy combinators help to combine strategies into more complex ones in a declaratively clear way. Transformations are nondeterministic and may yield several results, which fits very well into the logic programming paradigm. Strategic

* Partially funded by LIACC through Programa de Financiamento Plurianual of the Fundação para a Ciência e Tecnologia (FCT), by the FCT fellowship (ref. SFRH/BD/62058/2009), and by the EC FP6 for Integrated Infrastructures Initiatives under the project SCIEnce (contract No. 026133).

rewriting separates term traversal control from transformation rules. The separation of strategies and rules makes rules reusable in different transformations.

PρLog programs consist of clauses. The clauses either define user-constructed strategies by (conditional) transformation rules or are ordinary Prolog clauses. Prolog code can be used freely within PρLog programs, which is especially convenient when arithmetic calculations or input-output features are needed.

PρLog uses four different kinds of variables, which permits to traverse hedges in single/arbitrary width (with individual and sequence variables) and terms in single/arbitrary depth (with functional and context variables). It provides a possibility to extract an arbitrary subhedge from a hedge, or to extract subterms at arbitrary depth. In addition, PρLog permits regular constraints to restrict possible values of sequence and context variables by regular hedge expressions and regular tree (context) expressions, respectively. These constraints are very useful, for instance, in validation of an XML document with respect to a given DTD.

PρLog has not been implemented specifically for Web-related applications. Its main purpose is to bring strategy-based conditional hedge transformations in the logic programming framework for general programming. The role of PρLog in this paper is to provide a practical platform to illustrate suitability of the calculus behind it in XML querying, validation, and Web reasoning.

In the context of XML processing, the approach PρLog is based on can be classified as positional or pattern-based, where programmer specifies patterns including variables. Examples of such languages are Xcerpt [7], UnQL [8], XDuce [15], and CDuce [4]. Usually, in this approach, variables in patterns specify the nodes to be selected. With PρLog, we can select not only nodes but also sequences of nodes, node labels, and the context around a node that is at arbitrary depth. Moreover, it can naturally express incomplete query patterns.

Approaches to XML, based on the logic programming paradigm, have been quite popular. Besides the already mentioned Xcerpt, the languages like Elog [3], XPathLog [21], and XCentric [11, 12] belong to this category. The latter one, like PρLog, represents XML data as an unranked Prolog term and uses sequence matching with regular types for querying. In fact, for our experiments we used XCentric's XML-to-unranked-term translator.

From the general programming point of view, we should mention a number of calculi and languages for rule- and strategy-based programming related to our work, such as rewriting logic [20], ρ-calculus [9], ASF-SDF [25], CHR [14], ELAN [6], Maude [10], the OBJ family of languages [22], Stratego [26], and TOM [2]. ρLog, the calculus behind PρLog, has been influenced by the ρ-calculus. However, there are specific features in ρLog that makes it significantly different from the ρ-calculus: logic programming semantics, top-position matching, hedge transformations, different kinds of variables, and regular constraints.

2 PρLog

In this section we give a brief overview of basic features of PρLog, explaining them mostly on examples instead of giving formal definitions.

Terms and hedges (sequences of terms) in PρLog are built over unranked function symbols and four kinds of variables: individual, sequence, function, and context variables. These sets are disjoint. In this paper we follow the PρLog notation for this language, writing its constructs in `typewriter` font. PρLog uses the following conventions for the variables names: Individual variables start with `i_` (like, e.g., `i_Var` for a named variable or `i_` for the anonymous variable), sequence variables start with `s_`, function variables start with `f_`, and context variables start with `c_`. The function symbols, except the special constant `hole`, have flexible arity. To denote function symbols, PρLog basically follows the Prolog conventions for naming functors, operators, and numbers. Terms `t` and hedges `h` are constructed by the grammars:

```
t ::= i_X | f(h) | f_X(h) | c_X(t)    h ::= t | s_X | eps | (h_1, h_2)
```

where `eps` stands for the empty hedge and is omitted whenever it appears as a subhedge of another hedge. `a(eps)` and `f_X(eps)` are often abbreviated as `a` and `f_X`. A *Context* is a term with a single occurrence of `hole`. A context `C` can be applied to a term `t`, written `C[t]`, replacing the hole in `C` by `t`.

A *substitution* is a mapping from individual variables to hole-free terms, from sequence variables to hole-free hedges, from function variables to function variables/symbols, and from context variables to contexts, such that all but finitely many individual, sequence, and function variables are mapped to themselves, and all but finitely many context variables are mapped to themselves applied to the `hole`. This mapping is extended to terms/hedges in the standard way.

Matching problems are pairs of hedges, one of which is ground (i.e., does not contain variables). Such matching problems may have zero, one, or more (finitely many) solutions, called matching substitutions or *matchers*. For instance, the hedge `(s_1,f(i_X),s_2)` matches `(f(a),f(b),c)` in two different ways: one by the matcher `{s_1`$\mapsto$`eps,i_X`$\mapsto$`a,s_2`$\mapsto$`(f(b),c)}`, the other one by the matcher `{s_1`$\mapsto$`f(a),i_X`$\mapsto$`b,s_2`$\mapsto$`c}`. Similarly, the term `c_X(f_Y(a))` matches the term `f(a,g(a))` with the matchers `{c_X`$\mapsto$`f(hole,g(a)),f_Y`$\mapsto$`f}` and `{c_X`$\mapsto$`f(a,g(hole)),f_Y`$\mapsto$`g}`. An algorithm to solve matching problems in the described language has been introduced in [17].

Instantiations of sequence and context variables can be restricted by regular hedge and regular context languages, respectively, specified by the corresponding regular expressions. We do not go into the details of the regular constraints here, just mention that they can be added to matching problems to restrict the set of computed matchers, e.g., matching `c_X(f_Y(a))` to `f(a,g(a))` under the constraint `c_X in f(a,g(hole)*)` gives one matcher `{c_X`$\mapsto$`f(a,g(hole)),f_Y`$\mapsto$`g}` instead of two for the unconstrained case.

A ρLog *atom* (ρ-atom) is a quadruple consisting of a hole-free term `st` (a *strategy*), two hole-free hedges `h1` and `h2`, and a set of regular constraints `R` where each variable is constrained only once, written as `st :: h1 ==> h2 where R`. Intuitively, it means that the strategy `st` transforms `h1` to `h2` when the variables satisfy the constraint `R`. When `R` is empty, we omit it and write `st :: h1 ==> h2`. The negated atom is written as `st :: h1 =\=> h2 where R`. A ρLog *literal* (ρ-literal) is a ρ-atom or its negation. A PρLog *clause* is either a Prolog clause, or

a clause of the form `st :: h1 ==> h2 where R :- body` (in the sequel called a ρ-clause) where `body` is a (possibly empty) conjunction of ρ- and Prolog literals.

A PρLog *program* is a sequence of PρLog clauses. A *query* is a conjunction of ρ- and Prolog literals. ρ-clauses and queries can contain only ρLog variables. Prolog clauses and queries can contain only Prolog variables. If a Prolog literal occurs in a ρ-clause or query, it may contain only ρLog individual variables that internally get translated into Prolog variables.

PρLog inference mechanism is based essentially on SLDNF-resolution [1] adapted to ρ-clauses. If the selected literal in the query is a ρ-atom of the form `st :: h1 ==> h2 where R`, then PρLog finds a (renamed copy of a) ρ-clause `st' :: h1' ==> h2' where R' :- body` such that `st'` matches `st` and `h1'` matches `h1` with a substitution σ and the constraint `R'` is satisfied. Then, it replaces the selected literal in the query with the conjunction of `body`σ and a literal `id :: h2'`σ `==> h2 where R`, applies σ to the rest of the query and continues. `id` is a built-in strategy that forces `h2` to match `h2'`σ under `R`. To make sure that in this process we have matching and not unification, we impose *well-modedness* restrictions on ρ-clauses and queries. This is a quite technical notion, whose exact definition can be found in [19]. It guarantees that `h1` and `h2'`σ are ground. Negative ρ-literals are processed by the negation-as-failure rule.

3 XML Processing and Web Reasoning in PρLog

In this section, we illustrate how PρLog can be used in XML querying, validation, and reasoning. PρLog uses the unranked tree model, represented as a Prolog term. Below we assume that the XML input is provided in the translated form.

Querying. In [18], a list of query operations desirable for an XML query language is given: selection, extraction, reduction, restructuring, and combination. They all should be expressible in a single language. A comparison of five query languages on the basis of these queries can be found in [5]. Here we demonstrate, on an example, how selection, extraction, and reduction can be expressed in PρLog. The space limit does not allow us to illustrate the other queries.

Example 1. A car dealer office contains documents from different car dealers and brokers. There are two kinds of documents. The `manufacturer` documents list the manufacturer's name, year, and models with their names, front rating, side rating, and rank. The `vehicle` documents list the vendor, make, model, year, color and price. They are presented by XML data of the following form:

```
<manufacturer>
   <mn-name>Mercury</mn-name>
   <year>1998</year>
   <model>
     <mo-name>Sable LT</mo-name>
     <front-rating>3.84
     </front-rating>
```

```
<vehicle>
   <vendor>
      Scott Thomason
   </vendor>
   <make>Mercury</make>
   <model>Sable LT</model>
   <year>1999</year>
```

```
     <side-rating>2.14                  <color>
     </side-rating>                        metallic blue
     <rank>9</rank>                     </color>
   </model> ...                         <price>26800</price>
</manufacturer>                       </vehicle>
```

We assume that sequences of these elements are wrapped respectively by `<list-manuf>` and `<list-vehicle>` tags. To save space, in the queries below we use metavariable M to refer to the document with the root tag `<list-manuf>` and V to the document with the root tag `<list-vehicle>`.

Selection and Extraction: We want to select and extract `<manufacturer>` elements where some `<model>` has `<rank>` less or equal to 10.

```
select_and_extract :: list_manuf(s_,c_Manuf(rank(i_Rank)),s_) ==>
                      c_Manuf(rank(i_Rank)) :-
    i_Rank =< 10.
```

Given the goal `select_and_extract :: `M` ==> i_M`, this code generates all solutions, one after the other, via backtracking. The alternatives are generated according how `list_manuf(s_,c_Manuf(rank(i_Rank)), s_)` matches M.

Reduction: From the `<manufacturer>` elements, we want to drop the `<model>` subelements whose `<rank>` is greater than 10. Besides that, we also want to elide the `<front_rating>` and `<side_rating>` elements from the remaining models. It can be done in various ways in PρLog. One of such implementations is given below. `reduction` is defined as the normal form of transforming each `manufacturer` element inside `list_manuf`. A single `manufacturer` element is transformed by `reduction_step` depending whether it contains a model with the rank ≤ 10:

```
reduction :: list_manuf(s_1) ==> list_manuf(s_2) :-
    map1(nf(reduction_step)) :: s_1 ==> s_2,
    !.

reduction_step :: manufacturer(s_1,model(s_,rank(i_R)),s_2) ==>
                  manufacturer(s_1,s_2) :-
    i_R > 10.
reduction_step :: manufacturer(s_1,model(i_Name,i_,s_,rank(i_R)),s_2) ==>
                  manufacturer(s_1,model(i_Name,rank(i_R)),s_2) :-
    i_R =< 10.
```

Here `nf` is the built-in strategy for a normal form computation. Another built-in strategy, `map1`, maps its argument strategy to each single term of the input hedge. The query `reduction :: `M` ==> i_List` produces the list of reduced `manufacturer` elements.

Incomplete Queries. Often, the structure of a document to be queried is unknown to a query author, or she is interested not in the entire document but only in its relevant parts. A pattern-based Web querying language should

be able to express such incomplete queries. Schaffert in [24] classifies incomplete queries (four kinds of incompleteness: in breadth, in depth, with respect to order and with respect to optional elements) and explains how they are dealt with in Xcerpt. Here we show how they can be expressed in PρLog. As we will see, it can be done pretty naturally, without introducing additional constructs for them.

Incompleteness in breadth. In languages that have wildcards only for single terms, expressing incompleteness in breadth requires a special construct that allows to omit those wildcards for neighboring nodes in the data tree. In PρLog, we do not need any extra construct because of sequence variables. Anonymous sequence variables can be used as wildcards for arbitrary sequence of nodes. Furthermore, if needed, we can use named sequence variables to extract arbitrary sequence of nodes without knowing the exact structure. These are very convenient features, as one can see from the examples in the previous section.

Incompleteness in depth. It allows to select data items that are located at arbitrary, unknown depth and skip all structure in between. For this, in PρLog we just have to place the corresponding query subterm under an anonymous context variable. Moreover, if needed, we can extract the entire context above the query subterm without knowing the structure of the context, by putting there a named context variable. This has been done in the `select_and_extract` clause in the previous section with the `c_Manuf` variable. In fact, that clause also demonstrates how PρLog can combine incompleteness in breadth and depth in a single rule.

Incompleteness with respect to order. It allows to specify neighboring nodes in a different order than the one in that they occur in the data tree. Since PρLog does not permit matching in orderless theories,[4] we need a bit of more coding here. Assume that we do not know in which order the `front_rating` and `side_rating` elements occur in the model in Example 1 and write the clause that extract them:

```
extract_ratings :: c_(model(s_X)) ==> (i_Front,i_Side) :-
    id :: model(s_X) ==> model(s_1,front_rating(i_Front),s_2),
    id :: model(s_1,s_2) ==> model(s_,side_rating(i_Side),s_).
```

In the first subgoal of the body of this rule, the `id` strategy forces the term `model(s_1,front_rating(i_Front),s_2)` to match `model(s_X)`, extracting the value for front rating `i_Front`. Next, to find the side rating, we force matching `model(s_,side_rating(i_Side),s_)` to `model(s_1,s_2)` that is obtained from `model(s_X)` by deleting `front_rating(i_Front)`. This deletion comes for free from the previous match and we can take an advantage of it, since there is no need to keep `front_rating` in the structure where `side_rating` is looked for.

Incompleteness with respect to optional elements. Since seq. variables can be instantiated with the empty hedge, such queries are trivially expressed in PρLog.

[4] The orderless property is a generalization of commutativity for unranked function symbols. For orderless matching over unranked terms, see [16].

Validation. PρLog regular constraints can be used to check whether an XML document conforms to certain DTD that can be expressed by means of regular hedge expressions. We demonstrate it in the following example:

Example 2. Let the DTD below define the structure of the document containing manufacturer elements:

```
<!ELEMENT list-manuf (manufacturer*)>
<!ELEMENT manufacturer (mn-name, year, model*)>
<!ELEMENT model (mo-name, front-rating, side-rating, rank)>
<!ELEMENT mn-name (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT front-rating (#PCDATA)>
<!ELEMENT side-rating (#PCDATA)>
<!ELEMENT rank (#PCDATA)>
```

Then the validation task becomes a PρLog clause where DTD is encoded in a regular constraint:

```
validate :: s_X ==> true
   where [s_X in list_manuf(manufacturer(mn_name(i_),year(i_),
                               model(mo_name(i_),front_rating(i_),
                                 side_rating(i_),rank(i_))*)*)]
```

(With `i_` in the constraint we abbreviate the set of all ground terms with respect to the given finite alphabet.) To check whether a certain document conforms this DTD, we take a PρLog term T that represents that document and write the query `validate :: `T` ==> true`. The matching algorithm will try to match `s_X` to T and check whether the constraints are satisfied. If the document conforms the DTD, the query will succeed, otherwise it will fail.

Basic Web Reasoning. Semantic Web adds metadata to Web resources, which can be used to make retrieval "semantic". To query both data and metadata, languages need to have certain reasoning capabilities.

Example 3 (Clique of Friends, [24]). This example illustrates some basic reasoning (mainly the transitive closure of a relation) for the Semantic Web. It does not use any particular Semantic Web language itself.

Consider a collection of address books where each address book has an owner and a set of entries, some of which are marked as `friend` to indicate that the person associated with this entry is considered a friend by the owner of the address book. In XML, this collection of address books can be represented in a straightforward manner as follows:

```
<address-books>
  <address-book>                    <address-book>
    <owner>Donald Duck</owner>        <owner>Daisy Duck</owner>
    <entry>                           <entry>
      <name>Daisy Duck</name>           <name>Gladstone Duck</name>
      <friend/>                         <friend/>
```

```
    </entry>                              </entry>
    <entry>                               <entry>
      <name>Scrooge McDuck</name>           <name>Ratchet Gearloose</name>
                                            <friend/>
    </entry>                              </entry>
  </address-book>                       </address-book>
                                      </address-books>
```

The collection contains two address books, the first owned by `Donald Duck` and the second by `Daisy Duck`. Donalds address book has two entries, one for `Scrooge`, the other for `Daisy`, and only `Daisy` is marked as `friend`. Daisys address book again has two entries, both marked as `friend`.

The *clique-of-friends* of Donald is the set of all persons that are either direct friends of Donald (i.e. in the example above only `Daisy`) or friends of a friend (i.e. `Gladstone` and `Ratchet`), or friends of friends of friends (none in the example above), and so on. To retrieve these friends, we have to define the relation "being a friend of" and its transitive closure.

Transitive closure of a relation can be easily defined in PρLog. It can be even written in a generic way, parameterized by the strategy that defines the relation:

```
transitive_closure(i_Strategy) :: s_X ==> s_Y :-
    i_Strategy :: s_X ==> s_Y.
transitive_closure(i_Strategy) :: s_X ==> s_Z :-
    i_Strategy :: s_X ==> s_Y,
    transitive_closure(i_Strategy) :: s_Y ==> s_Z.
```

The relation of "being a friend of" with respect to the address books document is defined as follows:

```
friend_of(address_books(s_,
            address_book(owner(i_X),s_,entry(name(i_Y),friend),s_),
            s_)) :: i_X ==> i_Y.
```

The query `transitive_closure(friend_of(`T`)) :: Donald_Duck ==> i_Y`, where T is the PρLog term corresponding to the address book XML document above, will return one after the other the friend and the friends of the friend of `Donald_Duck:` `Daisy_Duck`, `Gladstone_Duck,` and `Ratchet_Gearloose.`

References

1. K. R. Apt and R. Bol. Logic programming and negation: A survey. *J. Logic Programming*, 19:9–71, 1994.
2. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Proc. RTA'07*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
3. R. Baumgartner, S. Flesca, and G. Gottlob. The Elog web extraction language. In *Proc. LPAR'01*, volume 2251 of *LNCS*, pages 548–560. Springer, 2001.
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proc. ICFP'03*, pages 51–63. ACM, 2003.

5. A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *ACM SIGMOD Record*, 29(1):68–79, 2000.
6. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. *ENTCS*, 4, 1996.
7. F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. ICLP'02*, number 2401 in LNCS. Springer, 2002.
8. P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.
9. H. Cirstea and C. Kirchner. The rewriting calculus - Parts I and II. *Logic Journal of the IGPL*, 9(3), 2001.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
11. J. Coelho and M. Florido. CLP(Flex): Constraint logic programming applied to XML processing. In *CoopIS, DOA, and ODBASE 2004*, volume 3291 of *LNCS*, pages 1098–1112. Springer, 2004.
12. J. Coelho and M. Florido. XCentric: A logic programming language for XML. Technical Report DCC-2005-X, DCC-FC and LIACC, University of Porto, 2005.
13. B. Dundua and T. Kutsia. PρLog. Version 0.7. Available from: `http://www.risc.uni-linz.ac.at/people/tkutsia/software.html`.
14. T. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3):95–138, 1998.
15. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
16. T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD thesis, Johannes Kepler University, Linz, 2002.
17. T. Kutsia and M. Marin. Matching with regular constraints. In *LPAR'05*, volume 3835 of *LNAI*, pages 215–229. Springer, 2005.
18. D. Maier. Database desiderata for and XML query language. Available from: `http://www.w3.org/TandS/QL/QL98/pp/maier.html`, 1998.
19. M. Marin and T. Kutsia. Foundations of the rule-based system RhoLog. *Journal of Applied Non-Classical Logics*, 16(1–2):151–168, 2006.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
21. W. May. XPath-Logic and XPathLog: a logic-programming-style XML data manipulation language. *TPLP*, 4(3):239–287, 2004.
22. The OBJ Family. `http://cseweb.ucsd.edu/~goguen/sys/obj.html`.
23. REWERSE. Reasoning on the web. `http://rewerse.net/`.
24. S. Schaffert. *Xcerpt: a rule-based query and transformation language for the Web*. PhD thesis, University of Munich, 2004.
25. M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. In *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.
26. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA'01*, volume 256 of *LNCS*, pages 357–362. Springer, 2001.