A Rule-Based Approach to XML Processing and Web Reasoning

Jorge Coelho[1], Besik Dundua[2], Mário Florido[3], and Temur Kutsia[4]

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

Abstract. We illustrate the potential of strategy-based 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. We show how to use it in XML querying, validation, and Web reasoning.

1 Introduction

The rule-based approach has been used extensively in many fields, such as expert systems, machine learning, theorem proving, tree automata, software building and configuration, program transformation, insurance and 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/transformation and Web reasoning as prominent examples. The REWERSE project [31] 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 in this paper is PρLog [18] (pronounced Pē-rō-log). It is a Prolog implementation of the ρLog calculus [27], 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 rewriting separates term traversal control from transformation rules. This allows the basic transformation steps to be defined concisely. 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. One can include its predicates in PρLog rules, which is especially convenient when arithmetic calculations or input-output features are needed.

PρLog uses four different kinds of variables in one framework, which allows to traverse hedges in single/arbitrary width (with individual and sequence variables) and terms in single/arbitrary depth (with functional and context variables). It facilitates flexibility in matching, providing a possibility to extract an arbitrary subhedge from a hedge, or to extract subterms at arbitrary depth. We illustrate these capabilities on the two examples, bubble sort and rewriting:

Example 1 (Bubble Sort).

```
swap :: (s_Subhedge1, i_X, s_Subhedge2, i_Y, s_Subhedge3) ==>
       (s_Subhedge1, i_Y, s_Subhedge2, i_X, s_Subhedge3) :-
    i_X @> i_Y.
```

The `swap` strategy looks in the input hedge for two terms `i_X` and `i_Y` (these are individual variables that stand for single terms) where `i_X` is larger in the standard order than `i_Y` (e.g., `5 @> 3` or `b @> a`) and swaps them. The sequence variables `s_Subhedge1, s_Subhedge2, s_Subhedge3` help to select such terms at arbitrary places as they stand for arbitrary subhedges of the input hedge. One can notice that swapping operation is implemented as a Prolog clause, in the form *Head*`:-`*Body*.

To implement bubble sort (in ascending order), we should keep swapping elements in the hedge until it is not possible anymore, i.e., we need to compute the normal form of the input hedge with respect to swapping. This can be easily done in PρLog with the built-in strategy `nf` for the normal form computation.

```
bubble_sort :: s_Input_Hedge ==> s_Sorted_Hedge :-
    nf(swap) :: s_Input_Hedge ==> s_Sorted_Hedge,
    !.
```

We put the cut at the end, because the same normal form can be computed in different ways and we are interested in only one result. One could also write a more general version of this program, where ordering is a parameter.

Example 2 (Rewriting). Implementation of term rewriting is straightforward:

```
rewrite(i_Rule) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
    i_Rule :: i_Redex ==> i_Contractum.
```

The context variable `c_Context` permits to descend in arbitrary depth in the input term, to find the redex `i_Redex` and construct the output term with the redex replaced by the contractum. Rewriting operation here is generic, parameterized with the rule `i_Rule` that performs the actual transformation of the redex. Which redex is selected (i.e., which rewriting strategy is applied), depends on how `c_Context` matches the input term. PρLog's built-in matching corresponds to leftmost-outermost term traversal, but we have shown in [19] that other rewriting strategies can be easily implemented inside PρLog.

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 a given XML document with respect to a given DTD.

It should be noted that 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. Usually, PρLog code is quite short, declaratively clear, and reusable. The user has also direct access to the control mechanism via strategies, that permits, for instance, the usage of clauses in different order for different tasks. 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 [8], UnQL [9], XML-QL [17], QBE [36], XDuce [21], and CDuce [5]. Usually, in the pattern-based 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. (Such queries have been introduced in [33].)

Approaches to XML, based on the logic programming paradigm, have been quite popular. Besides the already mentioned Xcerpt, there is XPathLog [29], an XPath-based declarative language with variable bindings that can used both for XML querying and for restructuring and integration of XML sources. Elog [4] is also a logic programming-based language for manipulation of XML data. XCentric [14, 15], 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.

The organization of the paper follows the goals we declared at the beginning: Section 2 describes PρLog. In Section 3 we illustrate capabilities of our rule-based approach on the examples of XML querying, incomplete querying, validation, and reasoning. Section 4 discusses some related work from the rule-based programming perspective.

2 PρLog

As it has already been mentioned in the introduction, PρLog is an implementation of the ρLog calculus in Prolog, extending the language with strategy-based conditional hedge transformation rules. 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`. For instance, applying the context `f(hole,b)` to `g(a)` gives `f(g(a),b)`.

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 and 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 can be extended to terms and hedges in the standard way. For instance, for a given substitution σ =`{c_Ctx`$\mapsto$`f(hole), i_Term`$\mapsto$`g(s_X),f_Funct`$\mapsto$`g,s_Hedge1`$\mapsto$`eps,s_Hedge2`$\mapsto$`(b,c)}` and a hedge `h=(c_Ctx(i_Term),f_Funct(s_Hedge1,a,s_Hedge2))`, we have that σ(`h`) = `(f(g(s_X)),g(a,b,c)).`

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`$\mapsto$`(),i_X`$\mapsto$`a,s_2`$\mapsto$`(f(b),c)}` and 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 [24].

Instantiations of sequence and context variables can be restricted by regular hedge and regular context languages, respectively. These constraints are expressed as `s_X in` RH and `c_X in` RC, where RH and RC are regular hedge and context expressions defined by the grammars:

$$\mathsf{RH} ::= \texttt{eps} \mid (\mathsf{RH}\ \mathsf{RH}) \mid \mathsf{RH}|\mathsf{RH} \mid \mathsf{RH}^* \mid \texttt{f}(\mathsf{RH}) \mid \mathsf{RC}(\texttt{f}(\mathsf{RH}))$$
$$\mathsf{RC} ::= \texttt{hole} \mid \mathsf{RC}.\mathsf{RC} \mid \mathsf{RC} + \mathsf{RC} \mid \mathsf{RC}^\star \mid \texttt{f}(\mathsf{RH}, \mathsf{RC}, \mathsf{RH})$$

For RH, juxtaposition stands for concatenation, the vertical bar $|$ for choice, and * for repetition. For RC, the dot is concatenation, $+$ is choice, and * is repetition. These expressions define the corresponding languages that are sets of ground hole-free hedges (for RH) and sets of ground contexts (for RC):

$$
\begin{aligned}
[\![\texttt{eps}]\!] &= \{\texttt{eps}\}.\\
[\![(\mathsf{RH}_1\ \mathsf{RH}_2)]\!] &= \{(\texttt{h1},\texttt{h2}) \mid \texttt{h1} \in [\![\mathsf{RH}_1]\!], \texttt{h2} \in [\![\mathsf{RH}_2]\!]\}.\\
[\![\mathsf{RH}_1|\mathsf{RH}_2]\!] &= [\![\mathsf{RH}_1]\!] \cup [\![\mathsf{RH}_2]\!].\\
[\![\mathsf{RH}^*]\!] &= \bigcup_{n\geq 0} [\![\mathsf{RH}]\!]^n.\\
[\![\texttt{f}(\mathsf{RH})]\!] &= \{\texttt{f}(\texttt{h}) \mid \texttt{h} \in [\![\mathsf{RH}]\!]\}.\\
[\![\mathsf{RC}(\texttt{f}(\mathsf{RH}))]\!] &= \{\texttt{C}[\texttt{f}(\texttt{h})] \mid \texttt{C} \in [\![\mathsf{RC}]\!], \texttt{h} \in [\![\mathsf{RH}]\!]\}.\\
[\![\texttt{hole}]\!] &= \{\texttt{hole}\}.\\
[\![\mathsf{RC}_1.\mathsf{RC}_2]\!] &= \{\texttt{C1}[\texttt{C2}] \mid \texttt{C1} \in [\![\mathsf{RC}_1]\!], \texttt{C2} \in [\![\mathsf{RC}_2]\!]\}.\\
[\![\mathsf{RC}_1 + \mathsf{RC}_2]\!] &= [\![\mathsf{RC}_1]\!] \cup [\![\mathsf{RC}_2]\!].\\
[\![\mathsf{RC}^\star]\!] &= \bigcup_{n\geq 0} [\![\mathsf{RC}]\!]^n.\\
[\![\texttt{f}(\mathsf{RH}_1, \mathsf{RC}, \mathsf{RH}_2)]\!] &= \{\texttt{f}(\texttt{h1}, \texttt{C}, \texttt{h2}) \mid \texttt{C} \in [\![\mathsf{RC}]\!], \texttt{h1} \in [\![\mathsf{RH}_1]\!], \texttt{h2} \in [\![\mathsf{RH}_2]\!]\}.
\end{aligned}
$$

Here $[\![\mathsf{RH}]\!]^0 = \{\texttt{eps}\}$, $[\![\mathsf{RH}]\!]^{n+1} = \{(\texttt{h1},\texttt{h2}) \mid \texttt{h1} \in [\![\mathsf{RH}]\!], \texttt{h1} \in [\![\mathsf{RH}]\!]^n\}$, $[\![\mathsf{RC}]\!]^0 = \{\texttt{hole}\}$, and $[\![\mathsf{RC}]\!]^{n+1} = \{\texttt{C1}[\texttt{C2}] \mid \texttt{C1} \in [\![\mathsf{RC}]\!], \texttt{C1} \in [\![\mathsf{RC}]\!]^n\}$ for $n > 0$.

We add regular constraints 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)*)`[5] gives one matcher `{c_X`$\mapsto$`f(a,g(hole)),f_Y`$\mapsto$`g}` instead of two for the unconstrained case mentioned earlier.

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`. We call `h1` the left hand side and `h2` the right hand side of this atom. 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 and a *query* is a conjunction of ρ- and Prolog literals. There is a restriction on variable occurrence imposed on clauses: ρ-clauses and queries can contain only ρLog variables, and 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 adapted to ρ-clauses. In these rules below, P stands for a program and Q denotes a query.

[5] Here we use simplified notation for regular expressions. The complete form would be `f(a(eps),g(eps,hole,eps)*,eps)`. It should also be noted that PρLog uses a bit different, more verbose syntax for regular operators, but we stick here to more conventional notation.

`id` is the built-in strategy for identity. The rules have the form $Q_1 \rightsquigarrow Q_2$, transforming the query Q_1 into a new query Q_2.

R: **Resolvent**

`st :: h1 ==> h2 where R` $\wedge\, Q \rightsquigarrow$
$\sigma($`body` $\wedge$ (`id :: h2' ==> h2 where R`) $\wedge\, Q)$

where `st` is not `id`, there exists a clause `st' :: h1' ==> h2' where R' :- body` in P such that under the constraint `R'`, the strategy `st'` matches `st` and the hedge `h1'` matches `h1` by the substitution σ.

Id: **Identity**

`id :: h1 ==> h2 where R` $\wedge\, Q \rightsquigarrow \sigma(Q)$

if under the constraint `R`, the hedge `h2` matches `h1` by the substitution σ.

NF: **Negation as Failure**

(`st :: h1 =\=> h2 where R`) $\wedge\, Q \rightsquigarrow Q$

if there exists a finitely failed SLDNF-derivation tree for `st :: h1 ==> h2 where R` with respect to P.

(We do not define here the standard notions like derivations, finitely failed SLDNF-derivation tree, etc. They can be found in the literature elsewhere, see, e.g, [1] for a survey). For Prolog clauses the usual SLDNF-resolution rules apply.

These rules can be applied in different (finitely many) ways to the same selected query and the same program clause, because there can be more than one matcher σ. But to guarantee that in derivations we face only matching problems and not unification problems (i.e., that the hedge `h1` in the rules above does not contain variables), we need to impose *well-modedness* restrictions on ρ-clauses and queries. This is a quite technical notion, whose definition can be found in [27] and which basically is based on the same notion for normal logic programs [16, 26, 2]. Roughly, the idea of well-modedness it to guarantee that whenever a ρ-atom is selected in the query, its left-hand side and the strategy term (input positions) do not contain uninstantiated variables. (For negative ρ-atoms this restriction extends to the right-hand sides as well, with the exception that anonymous variables are still permitted in the right-hand side.) This can be achieved if the variables in the input positions of a ρ-atom in a query occur also in the output positions (right-hand sides) of at least one of the ρ-literals located in the query to the left of that ρ-atom. (If the atom is in the body of a clause, than those variables may occur also in the left-hand side of the head of the clause.) For instance, the following clause is not well-moded, because `s_Y` occurs neither in the right-hand side of `str2 :: i_X ==> i_Y` nor in the left-hand side of `str1 :: (f(i_X),s_X) ==> (g(i_Z),s_Z)`:

```
str1 :: (f(i_X),s_X) ==> (g(i_Z),s_Z) :-
    str2 :: i_X ==> i_Y,
    str3 :: s_Y ==> s_Z.
```

In contrast, the clauses in Example 1 and Example 2 are well-moded. The main reason why we want to avoid unification is that first, it is infinitary (even with sequence variables only [23]), second, it subsumes context unification whose decidability is an open problem [32].

Strategies control rule applications. They can be either user-defined ones (like, e.g., `swap`, `bubble_sort`) or the built-in ones (e.g., `nf` and `id`). They can be ground or contain variables (e.g., as `rewrite(i_str)`), can be atomic or compound. PρLog comes with some predefined strategies, such as `compose` (sequential composition of its argument strategies), `choice` (nondeterministic choice), `map1` (maps its argument strategy to each single term of the input hedge), etc.

3 XML Processing and Web Reasoning in PρLog

In this section, we illustrate how PρLog can be used in XML querying, validation, and reasoning, pretty naturally and concisely expressing problems coming from these areas. For these applications, PρLog uses the unranked tree model, represented as a Prolog term. Below we assume that the XML input is provided in the translated form.

3.1 Querying

Maier in [25] gives a list of query operations that are desirable for an XML query language: selection, extraction, reduction, restructuring, and combination. They all should be expressible in a single query language. A comparison of five query languages on the basis of these queries is given in [6]. Here we demonstrate, on the car dealer office example, how these queries can be expressed in PρLog.

Example 3. 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>
     <side-rating>
        2.14
     </side-rating>
     <rank>9</rank>
   </model> ...
</manufacturer>
```

```
<vehicle>
   <vendor>
      Scott Thomason
   </vendor>
   <make>Mercury</make>
   <model>Sable LT</model>
   <year>1999</year>
   <color>
      metallic blue
   </color>
   <price>26800</price>
</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 consisting of the list of manufacturers, i.e., the document with the root tag `<list-manuf>`. Similarly, the metavariable V denotes 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 to the ways the term `list_manuf(s_,c_Manuf(rank(i_Rank)), s_)` matches M. If a `<manufacturer>` element contains two or more models with the rank ≤ 10, it will be returned several times. However, with a little modification of the code we can make sure that no such duplicated answers are computed:

```
select_and_extract :: list_manuf(s_,manufacturer(s_X),s_) ==>
                      manufacturer(s_X) :-
    select :: manufacturer(s_X) ==> manufacturer(s_X).

select :: c_Manuf(rank(i_Rank)) ==> c_Manuf(rank(i_Rank)) :-
    i_Rank =< 10,
    !.
```

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.
```

Then the query `reduction ::` M `==> i_List` produces the list of reduced `manufacturer` elements.

Join: We want our query to generate pairs of `<manufacturer>` and `<vehicle>` elements where `<mn-name>` = `<make>`, `<mo-name>` = `<model>` and `<year>` = `<year>`. The implementation is straightforward:

```
join :: (list_manuf(s_,
            manufacturer(mn_name(i_Manuf_Name),year(i_Year),
               s_,model(mo_name(i_Model_Name),s_X),s_),s_),
         list_vehicle(s_,
            vehicle(i_Vendor,make(i_Manuf_Name),
               model(i_Model_Name),year(i_Year),i_Price),s_)
        ) ==>
        (manufacturer(mn_name(i_Manuf_Name),year(i_Year),
            model(mo_name(i_Model_Name),s_X)),
         vehicle(i_Vendor,make(i_Manuf_Name),
            model(i_Model_Name),year(i_Year),i_Price)
        ).
```

The query `join ::` (M, V) `==> (i_Manuf,i_Vehicle)` returns a desired pair. One can compute all such pairs via backtracking.

Restructuring: We want our query to collect `<car>` elements listing their `make`, `model`, `vendor`, `rank`, and `price` subelements, in this order.

```
restructuring :: (list_manuf(s_,
                      manufacturer(mn_name(i_Manuf_Name),s_,
                          model(mo_name(i_Model_Name),s_X,i_Rank),
                          s_),
                      s_),
                  list_vehicle(s_,
                      vehicle(i_Vendor,make(i_Manuf_Name),
                          model(i_Model_Name),year(i_Year),
                          i_Color,i_Price),
                      s_)
                  )  ==>
                 car(make(i_Manuf_Name),model(i_Model_Name),
                      i_Vendor,i_Rank,i_Price).
```

The query `restructuring ::` (M, V) `==> i_Car` returns a car element. Backtracking gives all the answers.

3.2 Incomplete Queries

Often, the structure of a Web document to be queried is unknown to a query author, even if the schema to which the document conforms is familiar to her. The reason is that the schemas allow much flexibility for documents, expressed in terms of arbitrary repetition of substructures, or optional or alternative structures. Even more often, the query author is interested not in the entire structure of the document but only in its relevant parts. Therefore, a pattern-based Web querying language should be able to express such incomplete queries. Schaffert in [33] 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 the Xcerpt Language. 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. This is a very convenient feature. The second `select_and_extract` clause from the previous section is a good example. There the anonymous sequence variable `s_` helps to omit the irrelevant part of the document. The named sequence variable `s_X` helps to extract a sequence of nodes, without knowing its length and structure.

Incompleteness in depth. This kind of incompleteness allows to select data items that are located at arbitrary, unknown depth and skip all structure in between. The context variables in PρLog make this operation straightforward: Just place the query subterm you are interested in under an anonymous context variable. For instance, to extract only the rank values from the manufacturer elements in Example 3, we can write a simple clause:

```
select_rank :: c_(rank(i_Rank)) ==> i_Rank.
```

Here the anonymous context variable `c_` helps to descend to arbitrary depth, ignoring all the structure in between. We can do even more: If needed, we can extract the entire context above the query subterm without knowing the depth and the structure of the context. For this, it is enough just to put there a named context variable. This has been done in the first `select_and_extract` clause in the previous section with the `c_Manuf` variable:

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

In fact, this clause also demonstrates how incompleteness in breadth and depth can be combined in a single rule in PρLog.

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,[6] we need a bit of more coding to express incomplete queries with respect to order. For instance, assume that we do not know in which order the `front_rating` and `side_rating` elements occur in the model in Example 3 and write the clause that extract them:

[6] The orderless property is a generalization of commutativity for unranked function symbols. For orderless matching over unranked terms, see [22].

```
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 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. One can notice that in this example we also used incomplete queries in depth and breadth.

Incompleteness with respect to optional elements. It allows to query for certain substructures if they exist, but still let the query succeed if they do not exist. Since sequence variables can be instantiated with the empty sequence as well, such incomplete queries can be trivially expressed in PρLog.

3.3 Validation

PρLog permits regular constraints in its clauses for context and sequence variables. They, in particular, 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 4. 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.

Validation test can be tailored in XML transformations, to make sure that the result of the transformation conforms a given schema. Similar to the `validate` clause above, it is straightforward to express such tasks in PρLog. We do not elaborate on the details here.

3.4 Basic Web Reasoning

Reasoning plays a crucial role in making data processing on the Web more "intelligent". 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. In this section we demonstrate basic reasoning capabilities of PρLog using the Clique of Friends example from [33].

Example 5 (Clique of Friends, [33]). 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>
      <owner>Donald Duck</owner>
      <entry>
         <name>Daisy Duck</name>
         <friend/>
      </entry>
      <entry>
         <name>Scrooge McDuck</name>
      </entry>
   </address-book>
   <address-book>
      <owner>Daisy Duck</owner>
      <entry>
         <name>Gladstone Duck</name>
         <friend/>
      </entry>
      <entry>
         <name>Ratchet Gearloose</name>
         <friend/>
      </entry>
   </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.`

4 Related Work

In this paper, we discussed usability of strategic hedge transformations in Web-related applications and illustrated it with the rule-based tool PρLog. Although we have mentioned some of the systems related to PρLog in the introduction, the comparison made there was based on Web-related applications and not on general programming capabilities. From the latter point of view, there are a number of calculi and languages for rule- and strategy-based programming, such as rewriting logic [28], rewriting calculus [11], ASF-SDF [34], CHR [20], Claire [10], ELAN [7], Maude [13], the OBJ family of languages [30], Stratego [35], and TOM [3]. The ρLog calculus, on which PρLog is based on, has been influenced by the ρ-calculus that forms the theoretical basis of ELAN. However, there are specific features in ρLog that makes it significantly different from the ρ-calculus: logic programming semantics, top-position matching, hedge transformations, four different kinds of variables, and regular constraints.

PρLog design approach is close to the one behind the TOM language. With this approach, we make our technology available on top of an existing language. While we extend Prolog with strategies and hedge transformation rules, TOM extends Java with pattern matching capabilities and a strategy language. The list matching that TOM uses resembles very much to matching with sequence variables. However, TOM neither has context/function variables nor permits regular constraints on variables. The authors of the language indicate XML transformation as one of the applications TOM is especially suitable for [12].

CHR, like PρLog, extends Prolog in a declarative way. However, its purpose is different: It is a language for writing constraint solvers. PρLog is not designed specifically for programming constraint manipulations and we have not experimented with specifying such rules.

5 Acknowledgments

This research has been partially funded by LIACC through Programa de Financiamento Plurianual of the Fundação para a Ciência e Tecnologia (FCT), by the Georgia National Science Foundation (ref. YS09 2 1-120 and 09 184 1-120), and by the EC FP6 for Integrated Infrastructures Initiatives under the project SCIEnce (contract No. 026133).

References

1. K. R. Apt and R. Bol. Logic programming and negation: A survey. *J. Logic Programming*, 19:9–71, 1994.
2. K. R. Apt and S. Etalle. On the unification free Prolog programs. In *Proc. MFCS'93*, volume 711 of *LNCS*, pages 1–19. Springer, 1993.
3. 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.
4. 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.
5. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proc. ICFP'03*, pages 51–63. ACM, 2003.
6. A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *ACM SIGMOD Record*, 29(1):68–79, 2000.
7. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. *ENTCS*, 4, 1996.
8. 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.
9. 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.
10. Y. Caseau, F.-X. Josset, and F. Laburthe. Claire: combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming*, 2(6):769–805, 2002.
11. H. Cirstea and C. Kirchner. The rewriting calculus - Parts I and II. *Logic Journal of the IGPL*, 9(3), 2001.
12. H. Cirstea, P.-E. Moreau, and A. Reilles. TomML: A rule language for structured data. In *Proc. RuleML'09*, volume 5858 of *LNCS*, pages 262–271. Springer, 2009.

13. 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.
14. J. Coelho and M. Florido. CLP(Flex): Constraint logic programming applied to XML processing. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. Proc. of Confederated Int. Conferences*, volume 3291 of *LNCS*, pages 1098–1112. Springer, 2004.
15. 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.
16. P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *J. Logic Programming*, 2(2):119–155, 1985.
17. A. Deutsch, M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. XML-QL. In *QL*, 1998.
18. B. Dundua and T. Kutsia. PρLog. Version 0.7. Available from: `http://www.risc.uni-linz.ac.at/people/tkutsia/software.html`.
19. B. Dundua, T. Kutsia, and M. Marin. Strategies in PρLog. *EPTCS*, 15:32–43, 2010.
20. T. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3):95–138, 1998.
21. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
22. T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols.* PhD thesis, Johannes Kepler University, Linz, 2002.
23. T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symbolic Computation*, 42(3):352–388, 2007.
24. T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *Proc. LPAR'05*, volume 3835 of *LNAI*, pages 215–229. Springer, 2005.
25. D. Maier. Database desiderata for and XML query language. Available from: `http://www.w3.org/TandS/QL/QL98/pp/maier.html`, 1998.
26. J. Maluszynski and H. Komorowski. Unification-free execution of logic programs. In *Proc. 1985 Symposium on Logic Programming*, pages 78–86, 1985.
27. M. Marin and T. Kutsia. Foundations of the rule-based system RhoLog. *Journal of Applied Non-Classical Logics*, 16(1–2):151–168, 2006.
28. N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
29. W. May. XPath-Logic and XPathLog: a logic-programming-style XML data manipulation language. *TPLP*, 4(3):239–287, 2004.
30. The OBJ Family. `http://cseweb.ucsd.edu/~goguen/sys/obj.html`.
31. REWERSE. Reasoning on the web. `http://rewerse.net/`.
32. The RTA List of Open Problems. Problem #90. Available from the Web: `http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html`.
33. S. Schaffert. *Xcerpt: a rule-based query and transformation language for the web.* PhD thesis, University of Munich, 2004.
34. 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 R. Wilhelm, editor, *Proc. CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

35. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Proc. RTA'01*, volume 256 of *LNCS*, pages 357–362. Springer, 2001.
36. M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.