PρLog: Combining Logic Programming with Conditional Transformation Systems (Tool Description)*

Besik Dundua[1], Temur Kutsia[2], and Klaus Reisenberger-Hagmayer[3]

1 **Vekua Institute of Applied Mathematics, Tbilisi State University, Tbilisi, Georgia**
2 **RISC, Johannes Kepler University Linz, Linz, Austria**
3 **Johannes Kepler University Linz, Linz, Austria**

Abstract

PρLog extends Prolog by conditional transformations that are controlled by strategies. We give a brief overview of the tool and illustrate its capabilities.

1998 ACM Subject Classification D.1.6 Logic Programming, F.4.2 Grammars and Other Rewriting Systems, D.3.2 Language Classifications

Keywords and phrases Conditional transformation rules, strategies, Prolog

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.10

1 Brief overview

PρLog is a tool that combines, on the one hand, the power of logic programming and, on the other hand, flexibility of strategy-based conditional transformation systems. Its terms are built over function symbols without fixed arity, using four different kinds of variables: for individual terms, for sequences of terms, for function symbols, and for contexts. These variables help to traverse tree forms of expressions both in horizontal and vertical directions, in one or more steps. A powerful matching algorithm helps to replace several steps of recursive computations by pattern matching, which facilitates writing short and intuitively quite clear code. By the backtracking engine, nondeterministic computations are modeled naturally. Prolog's meta-programming capabilities allowed to easily write a compiler from PρLog programs (that consist of a specific Prolog code, actually) into pure Prolog programs.

PρLog program clauses either define user-constructed strategies by transformation rules or are ordinary Prolog clauses. Prolog code can be used freely in PρLog programs, which is especially convenient when built-ins, arithmetics, or input-output features are needed.

PρLog is based on the ρLog calculus [15], whose inference system is basically the SLDNF-resolution, with normal logic program semantics [14]. Therefore, Prolog was a natural choice to implement it. The ρLog calculus has been influenced by the ρ-calculus [5], which, in itself, is a foundation for the rule-based programming system ELAN [2]. There are some other languages for programming by rules, such as, e.g., ASF-SDF [16], CHR [11], Claire [4], Maude [6], Stratego [17], Tom [1]. The ρLog calculus and, consequently, PρLog differs from

* This research is partially supported by the Austrian Science Fund (FWF) under the projects P 24087-N18 and P 28789-N32, and by the Rustaveli National Science Foundation under the grants FR/508/4-120/14, FR/325/4-120/14, and YS15 2.1.2 70.

them, first of all, by its pattern matching capabilities. Besides, it adopts logic programming semantics (clauses are first class concepts, rules/strategies are expressed as clauses) and makes a heavy use of strategies to control transformations. We showed its applicability for XML transformation and Web reasoning [7], and in modeling rewriting strategies [9].

Here we briefly describe the current status of PρLog. A more detailed overview can be found in [10]. The system can be downloaded from its Web page `http://www.risc.jku.at/people/tkutsia/software/prholog/`. The current version has been tested for SWI-Prolog [18] version 7.2.3 or later.

2 How PρLog works

PρLog atoms are supposed to transform term sequences. Transformations are labeled by what we call *strategies*. Such labels (which themselves can be complex terms, not necessarily constant symbols) help to construct more complex transformations from simpler ones.

An instance of a transformation is finding duplicated elements in a sequence and removing one of them. We call this process double merging. The following strategy implements it:

$$\text{merge\_doubles} :: (s\_X,\ i\_x,\ s\_Y,\ i\_x,\ s\_Z) \Longrightarrow (s\_X,\ i\_x,\ s\_Y,\ s\_Z).$$

Here merge_doubles is the strategy name. It is followed by the separator :: which separates the strategy name from the transformation. Then comes the transformation itself in the form *lhs* $\Longrightarrow$ *rhs*. It says that if the sequence in *lhs* contains duplicates (expressed by two copies of the variable i_x, which can match individual terms and therefore, is called an *individual variable*) somewhere, then from these two copies only the first one should be kept in *rhs*. That "somewhere" is expressed by three sequence variables, where s_X stands for the subsequence of the sequence before the first occurrence of i_x, s_Y takes the subsequence between two occurrences of i_x, and s_Z matches the remaining part. These subsequences remain unchanged in the *rhs*. Note that one does not need to code the actual search process of doubles explicitly. The matching algorithm does the job instead, looking for an appropriate instantiation of the variables. There can be several such instantiations.

Now one can ask a question, e.g., to merge doubles in a sequence (1, 2, 3, 2, 1):

$$\text{?- merge\_doubles} :: (1,\ 2,\ 3,\ 2,\ 1) \Longrightarrow s\_Result.$$

PρLog returns two different substitutions: $\{s\_Result \mapsto (1, 2, 3, 2)\}$ and $\{s\_Result \mapsto (1, 2, 3, 1)\}$. They are computed via backtracking. Each of them is obtained from (1, 2, 3, 2, 1) by merging one pair of duplicates. A completely double-free sequence is just a normal form of this single-step transformation. PρLog has a built-in strategy for computing normal forms, denoted by **nf**, and we can use it to define a new strategy merge_all_doubles in the following clause (where :-, as in Prolog, stands for the inverse implication):

$$\text{merge\_all\_doubles} :: s\_X \Longrightarrow s\_Y \ \text{:-}\ \mathbf{nf}(\text{merge\_doubles}) :: s\_X \Longrightarrow s\_Y,\ !.$$

The effect of **nf** here is that it starts applying merge_doubles to s_X, and repeats this process iteratively as long as it is possible, i.e., as long as doubles can be merged in the obtained sequences. When merge_doubles is no more applicable, it means that the normal form of the transformation is reached and it is returned in s_Y. The Prolog cut at the end cuts the alternative ways of computing the same normal form. In general, Prolog primitives and clauses can be used freely in PρLog. Now, for the query

$$\text{?- merge\_all\_doubles} :: (1,\ 2,\ 3,\ 2,\ 1) \Longrightarrow s\_Result.$$

we get a single answer $s\_Result \mapsto (1, 2, 3)$. Instead of using the cut, we could have defined merge_all_doubles purely in PρLog terms, with the help of a built-in strategy **first_one**. It applies to a sequence of strategies (in the clause below there is only one such strategy, **nf**(merge_doubles)), finds the first one among them which successfully transforms the input sequence (s_X below), and gives back just *one result* of the transformation (in s_Y):

$$\text{merge\_all\_doubles} :: s\_X \Longrightarrow s\_Y \text{ :- } \mathbf{first\_one}(\mathbf{nf}(\text{merge\_doubles})) :: s\_X \Longrightarrow s\_Y.$$

PρLog is good not only in selecting arbitrarily many subexpressions in "horizontal direction" (by sequence variables), but also in working in "vertical direction", selecting subterms at arbitrary depth. *Context variables* provide this flexibility, by matching the context above the subterm to be selected. A context is a term with a single "hole" in it. When it applies to a term, the latter is "plugged in" the hole, replacing it. There is yet another kind of variable, called *function variable*, which stands for a function symbol. With the help of these constructs and the merge_doubles strategy, it is pretty easy to define a transformation that merges two identical branches in a tree, represented as a term:

$$\begin{array}{l}\text{merge\_double\_branches} :: c\_Con(f\_Fun(s\_X)) \Longrightarrow c\_Con(f\_Fun(s\_Y)) \text{ :- }\\ \quad \text{merge\_doubles} :: s\_X \Longrightarrow s\_Y.\end{array}$$

Here c_Con is a context variable and f_Fun is a function variable. This is a naming notation in PρLog, to start a variable name with the first letter of the kind of variable (*i*ndividual, *s*equence, *f*unction, *c*ontext), followed by the underscore. After the underscore, there comes the actual name. For anonymous variables, we write just $i\_$, $s\_$, $f\_$, $c\_$.

Now, we can ask to merge double branches in a given tree:

$$\text{?- merge\_double\_branches} :: f(g(a, b, a, h(c, c)), g(a, b, h(c))) \Longrightarrow i\_Result.$$

PρLog returns two different substitutions via backtracking:

$$\{i\_Result \mapsto f(g(a, b, h(c, c)), g(a, b, h(c)))\},$$
$$\{i\_Result \mapsto f(g(a, b, a, h(c)), g(a, b, h(c)))\}.$$

To obtain the first one, c_Con matched to the context $f(\circ, g(a, b, h(c)))$ (where $\circ$ is the hole), f_Fun to the symbol g, and s_X to the sequence $(a, b, a, h(c, c))$. merge_doubles transformed $(a, b, a, h(c, c))$ to $(a, b, h(c, c))$. The other result is obtained by matching c_Con to $f(g(a, b, a, \circ), g(a, b, h(c)))$, f_Fun to h, s_X to (c, c), and merging the c's in the latter.

One can have an arbitrary sequence (not necessarily a variable) in the right hand side of transformations in the queries, e.g., instead of i_Result above we could have had $c\_C(h(c, c))$, asking for the context of the result that contains $h(c, c)$. Then the output would be $\{c\_C \mapsto f(g(a, b, \circ), g(a, b, h(c)))\}$.

Similar to merging all doubles in a sequence above, we can also define a strategy that merges all identical branches in a tree repeatedly, as **first_one**(**nf**(merge_double_branches)). It would give $f(g(a, b, h(c)))$ for the input term $f(g(a, b, a, h(c, c)), g(a, b, h(c)))$.

PρLog execution principle is based on depth-first inference with leftmost literal selection in the goal. If the selected literal is a Prolog literal, then it is evaluated in the standard way. If it is a PρLog atom of the form $st :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$, due to the syntactic restriction called well-modedness (formally defined in [9]), st and $\tilde{s}_1$ do not contain variables. Then a (renamed copy of a) program clause $st' :: \tilde{s}'_1 \Longrightarrow \tilde{s}'_2$:- *body* is selected, such that st' matches st and $\tilde{s}'_1$ matches $\tilde{s}_1$ with a substitution σ. Next, the selected literal in the query is replaced with the conjunction $(body)\sigma$, **id** :: $\tilde{s}_2'\sigma \Longrightarrow \tilde{s}_2$, where **id** is the built-in strategy for identity: it succeeds iff its rhs matches the lhs. Evaluation continues further with this new query. Success and failure are defined in the standard way. Backtracking explores other alternatives that may come from matching the selected query literal to the head of the same program clause in a different way (since context/sequence matching is finitary, see, e.g., [8, 12, 13]), or to the head of another program clause. Negative literals are processed by negation-as-failure.

The PρLog distribution consists of the main file, parser, compiler, the library of built-in strategies, and a part responsible for matching. PρLog programs are written in files with the extension `.rho`. A PρLog session is initiated withing Prolog by consulting the main file. After that, the user can load a `.rho` file, which is parsed and compiled into a Prolog code. PρLog queries are also transformed into Prolog queries, which are then executed.

PρLog can be used in any development environment that is suitable for SWI-Prolog. We provide a special Emacs mode for PρLog, which extends the Prolog mode for Emacs [3]. It supports syntax highlighting, makes it easy to load PρLog programs and anonymize variables via the menu, etc. A tracing tool for PρLog is under development.

One can summarize the main advantages of PρLog as follows: compact and declarative code; capabilities of expression traversal without explicitly programming it; the ability to use clauses in a flexible order with the help of strategies. Besides, PρLog has access to the whole infrastructure of its underline Prolog system. These features make PρLog suitable for nondeterministic computations, manipulating XML documents, implementing rule-based algorithms and their control, etc.

References

1. Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on Java. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.
2. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan: A logical framework based on computational systems. *ENTCS*, 4, 1996.
3. Stefan D. Bruda. Prolog mode for Emacs (version 1.25), 2003. Available from `https://bruda.ca/emacs/prolog_mode_for_emacs`.
4. Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: combining sets, search and rules to better express algorithms. *TPLP*, 2(6):769–805, 2002.
5. Horatiu Cirstea and Claude Kirchner. The rewriting calculus - Parts I and II. *Logic Journal of the IGPL*, 9(3):339–410, 2001.
6. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.
7. Jorge Coelho, Besik Dundua, Mário Florido, and Temur Kutsia. A rule-based approach to XML processing and web reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR 2010*, volume 6333 of *LNCS*, pages 164–172. Springer, 2010.
8. Hubert Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998.
9. Besik Dundua, Temur Kutsia, and Mircea Marin. Strategies in PρLog. In Maribel Fernández, editor, *9th Int. Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009*, volume 15 of *EPTCS*, pages 32–43, 2009.
10. Besik Dundua, Temur Kutsia, and Klaus Reisenberger-Hagmayer. An overview of PρLog. RISC Report Series 16-05, Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria, 2016.

11 Thom W. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

12 Temur Kutsia. Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols. RISC Report Series 02-09, RISC, University of Linz, 2002. PhD Thesis.

13 Temur Kutsia and Mircea Marin. Matching with regular constraints. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

14 John Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

15 Mircea Marin and Temur Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

16 Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The Asf+Sdf meta-environment: a component-based language development environment. *Electr. Notes Theor. Comput. Sci.*, 44(2):3–8, 2001.

17 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.

18 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.