An Overview of PρLog

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

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

Abstract. This paper describes PρLog: a tool that combines Prolog with the ρLog calculus. Such a combination brings strategy-controlled conditional transformation rules into logic programming. They operate on sequences of terms. Transformations may lead to several results, which can be explored by backtracking. Strategies provide a control on rule applications in a declarative way. They are programmable: Users can construct complex strategies from simpler ones by special combinators. Different types of first- and second-order variables provide flexible control on selecting parts from sequences or terms. As a result, the obtained code is usually pretty compact and declaratively clear. In programs, PρLog-specific code can be intermixed with the standard Prolog code. The tool is implemented and tested in SWI-Prolog.

1 Introduction

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 within PρLog programs, which is especially convenient when some built-in primitives, arithmetic calculations, or input-output features are needed.

PρLog is based on the ρLog calculus [17] and, essentially, is its executable implementation. The inference system of the calculus is basically the SLDNF-resolution, with normal logic program semantics [15]. Therefore, Prolog was a natural choice to implement it.

Originally, the ρLog calculus evolved from experiments with extending the language of Mathematica [24] by a package for advanced rule-based programming [16,18]. Later, these developments influenced an extension of another symbolic computation system, Maple [19], by a rule-based programming package called symbtrans (an adaptation of ρLog) used for automatic derivation of multiscale models of arrays of micro- and nanosystems, see, e.g., [2].

The ρLog calculus has been influenced by the ρ-calculus [5], which, in itself, is a foundation for the rule-based programming system ELAN [3]. There are some other languages for programming by rules, such as ASF-SDF [21], CHR [12], Claire [4], Maude [6], Stratego [22], Tom [1], just to name a few. The ρLog calculus and, consequently, PρLog differs from 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. In earlier papers, we showed its applicability for XML transformation and Web reasoning [7], and in modeling rewriting strategies [10]. More recently, it has been used in extraction of frequent patterns from data mining workflows [20].

In this paper we describe the current status of the tool: Explain by simple examples how it works, discuss the language, architecture, built-in strategies, and the development environment. PρLog sources, Emacs mode, and help on built-in strategies 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 [23] version 7.2.3 or later.

2 Overview

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. Let us call this process double merging. The following strategy implements the idea:

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

The code, as one can see, is pretty short. 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 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, e.g., to merge doubles in a number sequence (1, 2, 3, 2, 1):

?- merge_doubles :: (1, 2, 3, 2, 1) $\Longrightarrow$ *s_Result*.

First, PρLog returns the substitution $\{s\_Result \mapsto (1, 2, 3, 2)\}$. Like in Prolog, the user may ask for more solutions, and, via backtracking, PρLog gives the second answer $\{s\_Result \mapsto (1, 2, 3, 1)\}$. Both are obtained from (1, 2, 3, 2, 1) by merging one pair of duplicates.

A double-free sequence is just a normal form of this single-step merge_doubles 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):

merge_all_doubles :: $s\_X \Longrightarrow s\_Y$:- **nf**(merge_doubles) :: $s\_X \Longrightarrow s\_Y$, !.

The effect of **nf** is that it applies merge_doubles to s_X, repeating 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. It is returned in s_Y.

Note the Prolog cut at the end. It cuts the alternative ways of computing the same normal form. In fact, Prolog primitives and clauses can be used in PρLog programs. Now, for the query

?- merge_all_doubles :: (1, 2, 3, 2, 1) $\Longrightarrow$ *s_Result*.

we get a single answer $s\_Result \mapsto (1, 2, 3)$.

Instead of the cut, we could define merge_all_doubles purely in PρLog terms:

merge_all_doubles :: $s\_X \Longrightarrow s\_Y$:-
    first_one(**nf**(merge_doubles)) :: $s\_X \Longrightarrow s\_Y$.

first_one is another PρLog built-in strategy. It applies to a sequence of strategies, finds the first one among them, which successfully transforms the input sequence, and gives back just *one result* of the transformation. Here it has a single argument strategy **nf**(merge_doubles) and returns (by instantiating s_Y) only one result of its application to s_X.

In the last clause, the transformation is exactly the same in the clause head and in the (singleton) body: Both transform s_X into s_Y. In such cases we can use more succinct notation:

merge_all_doubles := **first_one**(**nf**(merge_doubles)).

This form is called the *strategy definition* form: the strategy in its left hand side (here merge_all_doubles) is defined as the strategy in its right hand side (here **first_one**(**nf**(merge_doubles))).

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. Syntactically, the hole is denoted by a special constant. In the PρLog system it is `hole`, but here in the paper we use a more conventional notation $\circ$. 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 double branches in a tree, represented as a term:

merge_double_branches ::
$c\_Context(f\_Fun(s\_X)) \Longrightarrow c\_Context(f\_Fun(s\_Y))$:-
merge_doubles :: $s\_X \Longrightarrow s\_Y$.

Here $c\_Context$ 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:

?- merge_double_branches ::
$f(g(a,b,a,h(c,c)),\ h(c),\ g(a,a,b,h(c))) \Longrightarrow i\_Result.$

PρLog returns three results, one after the other, by backtracking:

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

To obtain the first one, PρLog matched the context variable $c\_Context$ to the context $f(\circ,\ h(c), g(a,a,b,h(c)))$, the function variable f_Fun to the function symbol g, and the sequence variable 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 results have been obtained by taking different contexts and respective subbranches.

The right hand side of transformations in the queries need not be variables. One can have an arbitrary sequence there. For instance, we may be interested in trees that contain $h(c,c)$:

?- merge_double_branches ::
$f(g(a,b,a,h(c,c)),\ h(c),\ g(a,a,b,h(c))) \Longrightarrow c\_C(h(c,c)).$

We get here two answers, which show instantiations of c_C by the relevant contexts:

$\{c\_C \mapsto f(g(a,b,\circ),\ h(c),\ g(a,a,b,h(c)))\},$

$\{c\_C \mapsto f(g(a,b,a,\circ),\ h(c),\ g(a,b,h(c)))\}$.

Similar to merging all doubles in a sequence above, we can also define a strategy that merges all identical branches in a tree repeatedly. It is not surprising that the built-in strategy for normal forms plays a role also here:

merge_all_double_branches := **first_one**(**nf**(merge_double_branches)).

For the query

?- merge_all_double_branches ::
$f(g(a,b,a,h(c,c)),\ h(c),\ g(a,a,b,h(c))) \Longrightarrow s\_Result$.

we get a single answer $\{s\_Result \mapsto f(g(a,b,h(c)),h(c))\}$.

Finally, note that a strategy can be defined by several clauses, which are treated as alternatives.

3 The PρLog Language

From the brief overview above one can get a pretty clear idea about the PρLog language: Its terms and sequences are constructed from function symbols that do not have fixed arity (variadic, aka unranked, function symbols), using the four kinds of variables. The constant hole is the exception: it is always used without the arguments. More precisely, terms are either individual variables, or expressions of one of the following forms: $f(\tilde{s})$, $f\_F(\tilde{s})$, or $c\_C(t)$, where f is an unranked function symbol, t is a term, and $\tilde{s}$ is a finite (possibly empty) sequence of terms or sequence variables. (These sequences are sometimes called hedges.) The empty sequence is denoted in the system with `eps`, but we use more conventional notation ( ) in the paper. Two sequences can be concatenated into one, where the empty sequence plays the role of the unit element of this (meta-level) concatenation operation. Sequences are written in the parenthesis for easy parsing (when they contain more than one element) and are flat. A singleton sequence is identified with its sole element. Contexts are terms with the unique occurrence of the hole. The previous section contains several examples of terms, sequences, and contexts.

Substitutions map individual variables to terms, sequence variables to sequences, function variables to function symbols or function variables, and context variables to contexts. For example, $\{c\_Ctx \mapsto f(\circ),\ i\_Term \mapsto g(s\_X),\ f\_Fun \mapsto g,\ s\_H_1 \mapsto (\,),\ s\_H_2 \mapsto (b,c)\}$ is a substitution. We can apply substitutions to sequences, which gives sequences as a result. In particular, if the sequence is a singleton term, then the result of the application is also a term. Applying the substitution above to a sequence $(c\_Ctx(i\_Term),\ f\_Fun(s\_H_1, a, s\_H_2))$ give the sequence $(f(g(s\_X)), g(a,b,c))$.

Note that sequence variables are not terms, and context variables always apply to terms, not to arbitrary sequences. This makes terms and contexts closed under substitution application.

The main computational mechanism for PρLog is matching. Due to sequence and context variables, it is finitary, which means that a matching problem may have finitely many solutions. For instance, the sequence $(s\_X,\ i\_x,\ s\_Y,\ i\_x,\ s\_Z)$ matches (1, 2, 3, 2, 1) in two different ways:

- $\{s\_X \mapsto (\,),\ i\_x \mapsto 1,\ s\_Y \mapsto (2,3,2),\ s\_Z \mapsto (\,)\}$,
- $\{s\_X \mapsto 1,\ i\_x \mapsto 2,\ s\_Y \mapsto 3,\ s\_Z \mapsto 1\}$.

(The parenthesis () above is the notation for the empty sequence used in this paper. PρLog uses `eps` for the ϵ-notation of it.)

In the previous section, we could also see two solutions to the problem of matching $c\_C(h(c,c))$ to the term $f(g(a,b,a,h(c,c)),h(c),\ g(a,a,b,h(c)))$.

A ρLog *atom* (ρ-atom) is a triple consisting of a strategy term st and two (hole-free) sequences $\tilde{s}_1$ and $\tilde{s}_2$, written as $st :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$. Its negation is written as $st :: \tilde{s}_1 \Longrightarrow\!\!\!\!\!/\ \ \tilde{s}_2$. 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 $st :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$:- *body* (called a ρ-*clause*) where *body* is a (possibly empty) conjunction of ρ- and Prolog literals. Strategy definitions $str_1 := str_2$ are shortcuts for clauses of the form $str_1 :: s\_X \Longrightarrow s\_Y$:- $str_2 :: s\_X \Longrightarrow s\_Y$.

In fact, PρLog clauses may have a more complex structure, when (some of) the literals are equipped with membership constraints, constraining possible values of sequence and context variables. Such constraints are taken into account in the matching process. For simplicity, we do not consider them in this paper.

A PρLog *program* is a sequence of PρLog clauses. A *query* is a conjunction of ρ- and Prolog literals. A restriction on variable occurrence is imposed on clauses: ρ-clauses and queries can contain only PρLog variables, while Prolog clauses and queries can contain only Prolog variables. If a ρ-clause or a query contains a Prolog literal, the only variables that can occur in that literal are PρLog individual variables. (When it comes to evaluating such Prolog literals, the individual variables there are converted into Prolog variables.)

PρLog programs have the extension `.rho`. In Fig. 1 one can see how exactly the program `merge.rho` for merging doubles in sequences and trees, discussed in Sect. 2, looks.

PρLog is implemented in SWI-Prolog, and its variables are, actually, Prolog constants. Therefore, one can not directly rely on Prolog unification to compute values for those variables. Consequently, the answers to the query should be computed as explicit substitutions showing what PρLog variables map to. It requires a PρLog query to be actually wrapped to a meta-query that then returns the substitutions. For the queries considered in Sect. 2, such meta-queries can be seen in Fig. 2. The predicate symbol used in them is `?`.

The substitutions indicate that there is a background solving mechanism in PρLog that performs matching and computes the corresponding substitutions. Indeed, we do it by the algorithm from [14], implemented in SWI-Prolog. However, it turns out that if we do not have context variables, then we can avoid using this implementation and, instead, compute matching substitutions directly by Prolog unification, which is, naturally, a more efficient way. We have imple-

```
% Merging double elements in a sequence:
% If the input sequence contains a double, keep the left copy.

merge_doubles :: (s_X, i_x, s_Y, i_x, s_Z) ==> (s_X, i_x, s_Y, s_Z).

% Merging all doubles:
% Return a normal form with respect to merge_doubles.

merge_all_doubles := first_one(nf(merge_doubles)).

% Merging double branches in a tree:
% If the input tree contains a double branch, keep the left one,
% using the merge_doubles strategy.

merge_double_branches ::
    c_Context(f_Fun(s_X)) ==> c_Context(f_Fun(s_Y)) :-
    merge_doubles :: s_X ==> s_Y.

% Merging all double branches in a tree:
% Return a normal form with respect to merge_double_branches.

merge_all_double_branches := first_one(nf(merge_double_branches)).
```

Fig. 1. Program `merge.rho` for merging doubles in sequences and trees.

mented this version of PρLog as well, calling it PρLog-light. To distinguish, we sometimes say PρLog-full for the version with context variables.

PρLog syntax in BNF notation can be found in the technical report [11] and on the system Web page.

We need to make sure that in the program execution process, all solving problems that arise for PρLog clauses and queries are matching problems, not unification. The reason is that matching for our language is finitary [14], while unification is infinitary [13,8]. The latter is undesirable, because it would cause infinite branching in the program execution tree. Therefore, we would like to restrict the solving to the fragment that guarantees an existence of a terminating finitary procedure. Matching is one of such possible fragments. The restriction we impose on clauses and queries is well-modedness, extends the same notion for logic programs, introduced in [9]. It forbids uninstantiated variables to appear in one of the sides of unification problems and, hence, only matching problems arise.

More specifically, well-modedness is based on the notion of mode of a relation. A mode for the relation $\cdot :: \cdot \Longrightarrow \cdot$ is a function that defines the input and output positions of the relation respectively as $in(\cdot :: \cdot \Longrightarrow \cdot) = \{1, 2\}$ and $out(\cdot :: \cdot \Longrightarrow \cdot) = \{3\}$. A mode is defined (uniquely) for a Prolog relation as well. A clause is moded if all its predicate symbols are moded. We assume that all ρ-clauses are moded. As for the Prolog clauses, we require modedness only for

```
?- ?(merge_doubles :: (1,2,3,2,1) ==> s_Result, Subst).
Subst = [s_Result---> (1, 2, 3, 2)] ;
Subst = [s_Result---> (1, 2, 3, 1)] ;
false.

?- ?(merge_all_doubles :: (1,2,3,2,1) ==> s_Result, Subst).
Subst = [s_Result---> (1, 2, 3)] ;
false.

?- ?(merge_double_branches ::
        f(g(a,b,a,h(c,c)), h(c), g(a,a,b,h(c))) ==> i_Result, Subst).
Subst = [i_Result--->f(g(a, b, h(c, c)), h(c), g(a, a, b, h(c)))] ;
Subst = [i_Result--->f(g(a, b, a, h(c)), h(c), g(a, a, b, h(c)))] ;
Subst = [i_Result--->f(g(a, b, a, h(c, c)), h(c), g(a, b, h(c)))] ;
false.

?- ?(merge_double_branches ::
        f(g(a,b,a,h(c,c)), h(c), g(a,a,b,h(c))) ==> c_C(h(c,c)), Subst).
Subst = [c_C--->f(g(a, b, hole), h(c), g(a, a, b, h(c)))] ;
Subst = [c_C--->f(g(a, b, a, hole), h(c), g(a, b, h(c)))] ;
false.

?- ?(merge_all_double_branches ::
        f(g(a,b,a,h(c,c)), h(c), g(a,a,b,h(c))) ==> i_Result, Subst).
Subst = [i_Result--->f(g(a, b, h(c)), h(c))] ;
false.
```

Fig. 2. Querying `merge.rho`.

those ones that define a predicate that occurs in the body of some ρ-clause. If a Prolog literal occurs in a query in conjunction with a ρ-clause, then its relation and the clauses that define this relation are also assumed to be moded.

Roughly, the idea of well-modedness is that the variables in the input positions should already be seen in the output positions of some earlier literals. Before defining it formally, we introduce the notation $vars(E)$ for a set of variables occurring in an expression E, and define $vars(E, \{p_1, ..., p_n\}) = \cup_{i=1}^{n} vars(E|_{p_i})$, where $E|_{p_i}$ is the standard notation for a subexpression of E at position p_i. The symbol $\mathcal{V}_\mathsf{a}$ stands for the set of anonymous variables. A ground expression contains no variables. Then well-moded queries and clauses are defined as follows:

Definition 1. *A* query $L_1, \ldots, L_n$ is well-moded *iff the following conditions hold for each* $1 \leq i \leq n$*:*

- $vars(L_i, in(L_i)) \subseteq \cup_{j=1}^{i-1} vars(L_j, out(L_j)) \setminus \mathcal{V}_\mathsf{a}$.
- *If* L_i *is a negative literal, then* $vars(L_i, out(L_i)) \subseteq \cup_{j=1}^{i-1} vars(L_j, out(L_j)) \cup \mathcal{V}_\mathsf{a}$.
- *If* L_i *is a* ρ*-literal, then its strategy term is ground.*

A clause L_0 :- $L_1, \ldots, L_n$ is well-moded *iff the following hold for each* $1 \leq i \leq n$*:*

- $vars(L_i, in(L_i)) \cup vars(L_0, out(L_0)) \subseteq \cup_{j=0}^{i-1} vars(L_j, out(L_j)) \setminus \mathcal{V}_{\mathtt{a}}$.
- *If* L_i *is a negative literal, then*

$$vars(L_i, out(L_i)) \subseteq \cup_{j=1}^{i-1} vars(L_j, out(L_j)) \cup \mathcal{V}_{\mathtt{a}} \cup vars(L_0, in(L_0)).$$

- *If* L_0 *and* L_i *are* ρ*-literals with the strategy terms* st_0 *and* st_i*, respectively, then* $vars(st_i) \subseteq vars(st_0)$.

It is easy to see that the clauses and queries in Fig. 1 and 2 are well-moded.

PρLog allows only well-moded program clauses and queries. There is no restriction on the Prolog clauses if the predicate they define is not used in a ρ-clause.

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 ρ-atom of the form $st :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$, the crucial thing is that, due to well-modedness, 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 the *rhs* matches the *lhs*. Evaluation continues further with this new query. Success and failure are defined in the standard way. Backtracking allows to explore other alternatives that may come from matching the selected query literal to the head of the same program clause in a different way, or to the head of another program clause. Negative literals are processed by the negation-as-failure rule. Well-modedness guarantees that whenever a negative ρ-literal is selected during the execution process, there are no variables in it except, maybe, some anonymous variables that may occur in its right-hand side.

4 System Components

The PρLog distribution consists of two parts: PρLog-full and PρLog-light. Each part has the main file, called `prholog.pl` and `prholog-l.pl`, respectively. They are responsible for setting up the environments and loading the corresponding version of PρLog. The major parts of both versions are the parser, compiler, and the library of built-in strategies: `parse.pl`, `compile.pl`, `library.pl` files for PρLog-full, and `parse-l.pl`, `compile-l.pl`, `library-l.pl` files for PρLog-light, respectively.

Besides, in the full PρLog there is a solver `solve.pl` for matching problems and regular constraints. The light version does not require such a solver, but it still needs to check regular constraints. It is done in the file `constraints-l.pl`.

A typical PρLog session starts by invoking SWI-Prolog and consulting the main PρLog file. After that, the user may write/edit a `.rho` file in her favorite editor, and load it by executing the query `?- load('...filename.rho')`, where

... stands for the full path. Next, the program can be queried as, e.g., it is shown in Fig. 2.

The parser and the compiler are invoked at the time when a `.rho` file is loaded. Besides syntax errors, the parser checks also for well-modedness and for occurrences of PρLog variables in Prolog literals. If no errors are detected, then the compiler compiles the `filename.rho` file into a Prolog file `filename.pl`, translating each PρLog clause into a Prolog clause. The file `filename.pl` is located in the same directory as `filename.rho`, loads immediately after the compilation, and is deleted on the exit.

For example, the full PρLog compiler compiles the clause that defines the `merge_all_doubles` strategy in Fig. 1 into the following Prolog clause:

```
merge_all_doubles(A, D, Q, S, T, V) :-
    solve([[hdg_rholog_internal]<<[hdg_rholog_internal|A]], B),
    instance([hdg_rholog_internal, seq_var(s_X)], B, C),
    solve([C<<D], E),
    ( append(B, E, F),
      instance([first_one, [nf, merge_doubles]], F, [G|I]),
      instance([hdg_rholog_internal, seq_var(s_X)], F, L),
      instance([hdg_rholog_internal, seq_var(s_Y)], F, M),
      instance([], F, N),
      get_strategy_name_arguments_from_the_list_form(G, J, H),
      append(H, I, K),
      O=..[J, K, L, M, N, F, P],
      O ),
    instance([hdg_rholog_internal, seq_var(s_Y)], P, R),
    solve([Q<<R|S], U),
    compose(T, U, V).
```

The constant `hdg_rholog_internal` is a tag for internal representation of sequences (hedges). The `solve` predicate is for solving matching problems. The predicate `instance` returns an instance of an expression under a substitution. `compose` composes substitutions. The univ predicate `=..` constructs the actual goal that is then evaluated. Since the clause is automatically generated, the variable names are not mnemonic and code formatting is rather non-standard.

The same parsing and compiling process is done when PρLog queries are evaluated. After compiling, the obtained Prolog query is executed. Answers are given as explicit substitutions.

5 Library

The library consists of definitions of built-in strategies, implemented in Prolog. They greatly simplify programming in PρLog. Currently there are 14 such strategies there. Except a couple of exception, each of them can be used both with and without regular constraints. We give a brief overview of some of those strategies, without mentioning the constraints.

Choice. The syntax of this strategy is

$$\textbf{choice}(\mathit{strategy}_1, \ldots, \mathit{strategy}_n) :: \mathit{sequence}_1 \Longrightarrow \mathit{sequence}_2,$$

where $n \geq 1$. It succeeds if and only if for some i, $\mathit{strategy}_i :: \mathit{sequence}_1 \Longrightarrow \mathit{sequence}_2$ succeeds.

Composition. Composing strategies, making the output sequence of one the input for the other:

$$\textbf{compose}(\mathit{strategy}_1, \ldots, \mathit{strategy}_n) :: \mathit{sequence}_1 \Longrightarrow \mathit{sequence}_2,$$

where $n \geq 2$. First applies $\mathit{strategy}_1$ to $\mathit{sequence}_1$. To its result, $\mathit{strategy}_2$ is applied and so on. $\mathit{sequence}_2$ is the final result. **compose** fails if one of its argument strategies fails in the process.

Closure. The syntax of this strategy is:

$$\textbf{closure}(\mathit{strategy}) :: \mathit{sequence}_1 \Longrightarrow \mathit{sequence}_2,$$

It succeeds if $\mathit{sequence}_2$ belongs to the closure set of transforming $\mathit{sequence}_1$ by *strategy*. The set elements are computed one after the other, by backtracking. **closure** fails if the set is empty. An example of a query would be

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

It gives five answer substitutions via backtracking:

- $\{\mathit{s\_Result} \mapsto (1, 2, 3, 2, 1)\}$,
- $\{\mathit{s\_Result} \mapsto (1, 2, 3, 2)\}$,
- $\{\mathit{s\_Result} \mapsto (1, 2, 3)\}$,
- $\{\mathit{s\_Result} \mapsto (1, 2, 3, 1)\}$,
- $\{\mathit{s\_Result} \mapsto (1, 2, 3)\}$

Identity. The goal of this strategy is to transform a sequence to its identical one:

$$\textbf{id} :: \mathit{sequence}_1 \Longrightarrow \mathit{sequence}_2.$$

It succeeds iff $\mathit{sequence}_2$ can match $\mathit{sequence}_1$.

Returning all answers of the first applicable strategy, one by one. Denoted by **first_all**:

$$\textbf{first\_all}(\mathit{strategy}_1, \ldots, \mathit{strategy}_n) :: \mathit{sequence}_1 \Longrightarrow \mathit{sequence}_2,$$

where $n \geq 1$. Tries to apply $\mathit{strategy}_1$ to $\mathit{sequence}_1$. If this fails, it tries the next strategy and so on. When a strategy is found that succeeds, **first_all** returns *all answers* computed by it in $\mathit{sequence}_2$, via backtracking. If no strategy succeeds, **first_all** fails.

The strategy **first_one** mentioned earlier is similar to **first_all**, with the only difference that it returns only one answer instead of all of them.

Returning all answers at once. It can be seen as an analog of `findall` for PρLog. The syntax is

all_answers(*strategy*) :: $sequence_1 \Longrightarrow sequence_2$.

It succeeds if and only if $sequence_2$ is a sequence consisting of terms of the form $ans(\tilde{s}_1), \ldots, ans(\tilde{s}_n)$, where $\tilde{s}_1, \ldots, \tilde{s}_n$ are all the sequences obtained by applying *strategy* to $sequence_1$. The symbol *ans* just plays the role of a constructor, to distinguish between different answer sequences in $sequence_2$. We could ask

$? -$ **all_answers**(*merge_doubles*) :: (1, 2, 3, 2, 1) $\Longrightarrow$ *s_Result*.

and obtain the answer $\{s\_Result \mapsto (ans(1, 2, 3, 2), ans(1, 2, 3, 1))\}$.

Interactive mode. The syntax is:

interactive :: $sequence_1 \Longrightarrow sequence_2$.

It activates the interactive mode and starts dialog with the user, asking her to provide a strategy, which is then applied to $sequence_1$. The process is repeated further so that the output sequence of the previous strategy application becomes the input for the new strategy provided by the user, and so on. The interactive process stops when the user types *finish.* At that moment, the input sequence that was there is returned in $sequence_2$. **interactive** fails when the user-provided strategy fails for the current input sequence.

n-fold iteration. Specifies how many times a strategy can be applied repeatedly:

iterate(*strategy*, n) :: $sequence_1 \Longrightarrow sequence_2$.

It applies *strategy* repeatedly, n times, starting from $sequence_1$. The result is returned in $sequence_2$. **iterate** fails if one of the applications fails.

The normal form strategy **nf** is similar, but instead of applying a strategy fixed number of times, it applies it until the transformation is not possible, and returns the last sequence.

Mapping a strategy to a sequence. Mapping is a common operation in declarative programming:

map(*strategy*) :: $sequence_1 \Longrightarrow sequence_2$.

It applies *strategy* to each term of $sequence_1$. For such an input term, *strategy* may, in general, return a sequence (not necessarily a single term). A sequence constructed of these results (in the same order) is then returned in $sequence_2$. **map** fails when the application of *strategy* to a term from $sequence_1$ fails. When $sequence_1$ is empty, $sequence_2$ is empty as well.

A variation of this strategy, **map_to_subhedges**, splits $sequence_1$ nondeterministically into nonempty subsequences (when $sequence_1$ is not empty) and applies *strategy* to each such subsequence. A sequence constructed from these results (in the same order) is returned in $sequence_2$. **map_to_subhedges** fails when $sequence_1$ can not split in such a way that the application of *strategy* succeeds for each split subsequence. When $sequence_1$ is empty, so is $sequence_2$.

Rewriting. Yet another common transformation, which transforms a term not necessarily in the top position, but by transforming its subterm, in general:

$$\textbf{rewrite}(\mathit{strategy}) :: \mathit{term}_1 \Longrightarrow \mathit{term}_2.$$

It succeeds if and only if term_2 is obtained from term_1 by applying *strategy* to a subterm of it. Note that one can easily define rewriting inside full PρLog:

$$\mathit{rewrite}(\mathit{i\_Strategy}) :: \mathit{c\_Context}(\mathit{i\_Term}_1) \Longrightarrow \mathit{c\_Context}(\mathit{i\_Term}_2) \text{ :-}$$
$$\quad \mathit{i\_Strategy} :: \mathit{i\_Term}_1 \Longrightarrow \mathit{i\_Term}_2.$$

Nevertheless, we decided to provide the predefined strategy for rewriting in the library, because it is quite a frequently used transformation.

6 Development Environment

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 Stefan D. Bruda's Prolog mode for Emacs.[4] It supports syntax highlighting, makes it easy to load PρLog programs and anonymize variables via the menu, etc. Fig. 3 can give an idea how it looks.

A tracing tool for PρLog is under development. Prolog trace is too fine-grained for this purpose, since it goes through all parsing and compilation predicates that are invoked when a PρLog query is evaluated. Instead, the PρLog-specific tracing/debugging tool should ignore (by default) all intermediate Prolog steps and show only those that are directly related to PρLog inference.

7 Discussion and Final Remarks

The main advantage of using PρLog is its flexibility in specifying nondeterministic computations, which allows to neatly combine conditional transformation rules with logic programming. Strategies help to separate transformation rules from the control on their application, which makes rules reusable in different transformations. It also means that, unlike Prolog, the user can apply the program clauses in different order for different queries, without rewriting the code.

Assume that we have two PρLog rules, one for the top-level transformation of a term, and the other one for transforming an argument:

$$\mathit{transform\_top}(\mathit{i\_Strategy}) :: \mathit{i\_Term}_1 \Longrightarrow \mathit{i\_Term}_2 \text{ :-}$$
$$\quad \mathit{i\_Strategy} :: \mathit{i\_Term}_1 \Longrightarrow \mathit{i\_Term}_2.$$
$$\mathit{transform\_arg}(\mathit{i\_Strategy}) ::$$
$$\quad \mathit{f\_Fun}(\mathit{s\_X}, \mathit{i\_Term}_1, \mathit{s\_Y}) \Longrightarrow \mathit{f\_Fun}(\mathit{s\_X}, \mathit{i\_Term}_2, \mathit{s\_Y}) \text{ :-}$$
$$\quad \mathit{i\_Strategy} :: \mathit{i\_Term}_1 \Longrightarrow \mathit{i\_Term}_2.$$

[4] `https://bruda.ca/emacs/prolog_mode_for_emacs`

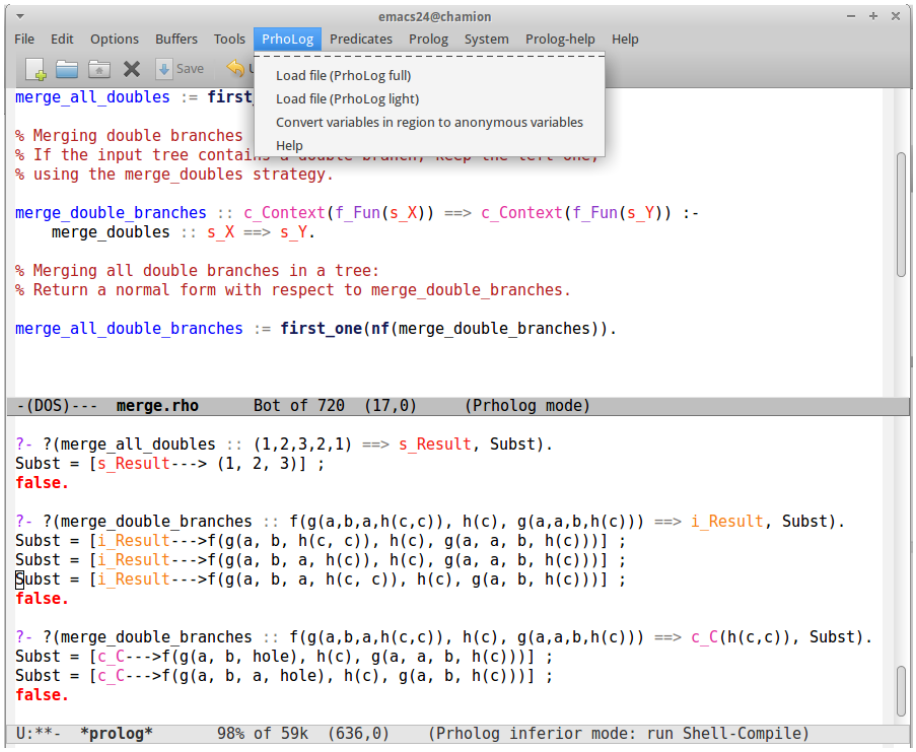

Fig. 3. Emacs PρLog session.

Note that the use of function and sequence variables makes the code universal (it can apply to any term, independent to their top function symbols and the number of arguments) and compact (one does not need to implement the term decomposition and traversal explicitly, the declarative specification given above is sufficient).

Now, innermost and outermost rewriting strategies can be implemented by strategy combinations only, imposing the right application order of the transformation rules.

Innermost rewriting is defined by the following recursive strategy:

$$\begin{aligned} &innermost\_rewriting(i\_Strategy) := \\ &\quad \textbf{first\_all}(transform\_arg(innermost\_rewriting(i\_Strategy)), \\ &\qquad transform\_top(i\_Strategy)). \end{aligned}$$

It gives the priority to the argument transformation by innermost rewriting (wrt the given strategy) over the top-position transformation (wrt the given strategy): If the former is applicable, **first_all** makes sure that its all possible results are returned and the latter is not tried. For instance, assume that *str* is

some concrete strategy defined by two clauses:

$$str :: f(s\_X) \Longrightarrow g(s\_X). \qquad\qquad str :: f(f(i\_X)) \Longrightarrow i\_X.$$

If we ask to rewrite $h(f(f(a)), f(a))$ by innermost rewriting:

$$? - \ innermost\_rewriting(str) :: h(f(f(a)), f(a)) \Longrightarrow i\_Result.$$

PρLog will return two results: $h(f(g(a)), f(a))$ and $h(f(f(a)), g(a))$.

If we want to experiment with outermost rewriting, we only need to define the corresponding strategy (essentially, by changing the application order of the rules, without altering them):

$$\begin{array}{l} outermost\_rewriting(i\_Strategy) := \\ \quad \textbf{first\_all}(transform\_top(i\_Strategy), \\ \qquad transform\_arg(outermost\_rewriting(i\_Strategy))). \end{array}$$

Rewriting $h(f(f(a)), f(a))$ by this strategy gives three results: $h(g(f(a)), f(a))$, $h(a, f(a))$, and $h(f(f(a)), g(a))$.

The definitions also clearly illustrate the difference between these two rewriting strategies.

If one wants to compute only one result, instead of all, the only change needed in this case is to replace **first_all** by **first_one** in the corresponding strategy.

This example shows some advantages of PρLog: 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, implementing rule-based algorithms and their control, manipulating XML documents, etc.

As future work, one direction is finishing the implementation of PρLog trace. We also plan to improve the compiler by adding more optimization capabilities.

Acknowledgments

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 (GSRNSF) under the grants FR/508/4-120/14, FR/325/4-120/14 and YS15 2.1.2 70.

References

1. E. Balland, P. Brauner, R. Kopetz, P. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In F. Baader, editor, *RTA 2007*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
2. W. Belkhir, A. Giorgetti, and M. Lenczner. A symbolic transformation language and its application to a multiscale method. *J. Symb. Comput.*, 65:49–78, 2014.

3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. *ENTCS*, 4, 1996.
4. Y. Caseau, F. Josset, and F. Laburthe. CLAIRE: combining sets, search and rules to better express algorithms. *TPLP*, 2(6):769–805, 2002.
5. H. Cirstea and C. Kirchner. The rewriting calculus - Parts I and II. *Logic Journal of the IGPL*, 9(3):339–410, 2001.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.
7. J. Coelho, B. Dundua, M. Florido, and T. Kutsia. A rule-based approach to XML processing and web reasoning. In P. Hitzler and T. Lukasiewicz, editors, *RR 2010*, volume 6333 of *LNCS*, pages 164–172. Springer, 2010.
8. H. Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998.
9. P. Dembinski and J. Maluszynski. And-parallelism with intelligent backtracking for annotated logic programs. In *Proc. 1985 Symposium on Logic Programming*, pages 29–38. IEEE-CS, 1985.
10. B. Dundua, T. Kutsia, and M. Marin. Strategies in PρLog. In M. Fernández, editor, *9th Int. Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009*, volume 15 of *EPTCS*, pages 32–43, 2009.
11. B. Dundua, T. Kutsia, and K. Reisenberger-Hagmayer. An overview of PρLog. RISC Report Series 16-05, RISC, University of Linz, 2016.
12. T. W. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
13. T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.
14. T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *LNCS*, pages 215–229. Springer, 2005.
15. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
16. M. Marin and T. Kutsia. On the implementation of a rule-based programming system and some of its applications. In B. Konev and R. Schmidt, editors, *Proc. 4th Int. Workshop on the Implementation of Logics, WIL'04*, pages 55–69, 2003.
17. M. Marin and T. Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.
18. M. Marin and F. Piroi. Rule-based programming with Mathematica. In *Proceedings of the 6th International Mathematica Symposium, Alberta, Canada*, 2004.
19. M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005.
20. P. Nguyen. *Meta-mining: a meta-learning framework to support the recommendation, planning and optimization of data mining workflows*. PhD thesis, Department of Computer Science, University of Geneva, 2015.
21. M. van den Brand, A. van Deursen, J. Heering, H. 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. *Electr. Notes Theor. Comput. Sci.*, 44(2):3–8, 2001.
22. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *RTA 2001*, volume 2051 of *LNCS*, pages 357–362. Springer, 2001.
23. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
24. S. Wolfram. *The Mathematica book (5. ed.)*. Wolfram-Media, 2003.