Strategies in PρLog

Besik Dundua
RISC, JKU Linz, Austria
bdundua@risc.uni-linz.ac.at

Temur Kutsia
RISC, JKU Linz, Austria
kutsia@risc.uni-linz.ac.at

Mircea Marin
University of Tsukuba, Japan
mmarin@cs.tsukuba.ac.jp

PρLog is an experimental extension of logic programming with strategic conditional transformation rules, combining Prolog with ρLog calculus. The rules perform nondeterministic transformations on hedges. Strategies provide a control on rule applications in a declarative way. In this short paper we give an overview on programming with strategies in PρLog and demonstrate how rewriting strategies can be expressed.

1 Introduction

PρLog (pronounced Pē-rō-log) is an experimental tool that extends logic programming with strategic conditional transformation rules, combining Prolog with ρLog calculus [9]. ρLog deals with hedges, transforming them by conditional rules. Transformations are nondeterministic and may yield several results. Strategies provide a control on rule applications in a declarative way. The rules apply matching to the whole input hedge (or, if it is a single term, apply at the top position). Four different types of variables give the user flexible control on selecting terms in hedges (via individual and sequence variables) or subterms in terms (via function and context variables). As a result, the obtained code is usually quite short and declaratively clear. We tried to provide as little as possible hard-wired features[1] to give the user a freedom in experimenting with different choices.

PρLog inference mechanism is essentially the same as SLDNF-resolution, multiple results are generated via backtracking, its semantics is compatible with semantics of normal logic programs [8] and, hence, Prolog was a natural choice to base PρLog on: The inference mechanism comes for free, as well as the built-in arithmetic and many other useful features of the Prolog language. Prolog code can be used freely within PρLog programs. Following Prolog, PρLog is also untyped, but values of sequence and context variables can be constrained by regular hedge or tree languages. We do not elaborate on this feature here.

Programming with rules has been experiencing a period of growing interest since the nineties when rewriting logic [10] and rewriting calculus [3] have been developed and several systems and languages (ASF-SDF [12], CHR [6], Claire [2], ELAN [1], Maude [4], Stratego [13], just to name a few) emerged. The ρLog calculus has been influenced by the ρ-calculus [3] as also its name suggests, but there are some significant differences: ρLog adopts logic programming semantics (clauses are first class concepts, rules/strategies are expressed as clauses), uses top-position matching, and employs four different kinds of variables. Consequently, PρLog (based on ρLog) differs from ELAN (based on ρ-calculus). Also, ELAN is a mature system with a very efficient compiler while PρLog is an experimental extension of Prolog implemented in Prolog itself. From the architecture point of view, PρLog is closer to another mature system, CHR, because both extend the host language (in this case, Prolog) in a declarative way. CHR extends it with the rules to handle constraints that are the first class concept there.

[1] Probably the most notable such feature is the leftmost-outermost term traversal strategy the matching algorithm uses, but it can also be easily modified since the corresponding Prolog code is open: Exchanging the order of clauses would suffice. The user can also program different traversal strategies pretty easily inside PρLog.

This is a preliminary version of a paper
that will appear in Electronic Proceedings
in Theoretical Computer Science.

The goal of this short paper is to give an overview of PρLog and, in particular, show how it uses strategies. First, we discuss syntax of PρLog (Sect. 2), then list some of the strategies from the library with a brief explanation (Sect. 3), and show how user-defined strategies can be introduced. We illustrate this on the examples of defining rewriting strategies in PρLog (Sect. 4). One can see that the code there is quite short and readable, and it also demonstrates expressiveness of PρLog.

2 Preliminaries

PρLog is essentially based on the language of ρLog [9], extending Prolog with it. Here we use the PρLog notation for this language, writing its constructs in `typewriter` font. The expressions are built over the set of functions symbols $\mathscr{F}$ and the sets of individual, sequence, function, and context variables. These sets are disjoint. 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 symbols in $\mathscr{F}$, except the special constant `hole`, have flexible arity. To denote the symbols in $\mathscr{F}$, PρLog basically follows the Prolog conventions for naming functors, operators, and numbers.

Terms `t` and hedges `h` are constructed in a standard way: `t` ::= `i_X` | `hole` | `f(h)` | `f_X(h)` | `c_X(t)` and `h` ::= `t` | `s_X` | `eps` | $\texttt{h}_1$,$\texttt{h}_2$, 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`. Context is a term with a single occurrence of `hole`. A context can be applied to a term, replacing the hole by that term. Substitution maps individual variables to (hole-free) terms, sequence variables to (hole-free) hedges, function variables to function symbols, and context variables to contexts (all but finitely many individual, sequence, and function variables are mapped to themselves, all but finitely many context variables are mapped to themselves applied to the hole). In [7], an algorithm to solve matching equations in the language just described has been introduced. Two hedges may have zero, one, or more (finitely many) matchers.

A ρLog atom (ρ-atom) is a triple consisting of a term `st` (a strategy) and two hole-free hedges `h1` and `h2`, written as `st :: h1 ==> h2`. Intuitively, it means that the strategy `st` transforms the hedge `h1` to the hedge `h2`. (We will use this, somehow sloppy, but intuitively clear wording in this paper.) Its negation is written as `st :: h1 =\=> h2`. A ρLog 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 :- 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. A restriction imposed on clause and queries is that no ρLog variables can occur in Prolog clauses and no Prolog variables occur in ρ-clauses. When a Prolog literal appears in a body of a ρ-clause, or in a query with a ρ-literal, then the only variables that the Prolog literal may contain are the ρLog individual variables. (When it comes to evaluating such Prolog literals, the individual variables there are converted into Prolog variables.)

Both a program clause and a query should satisfy a syntactic restriction, called well-modedness, to guarantee that each execution step is performed using matching (which is finitary in our language) and not unification (whose decidability is not known[2]). Well-modedness for PρLog programs extends the same notion for logic programs [5]: A mode for the relation $\cdot$ `::` $\cdot$ `==>` $\cdot$ is a function that defines the input and output positions respectively as $in(\cdot \texttt{ :: } \cdot \texttt{ ==> } \cdot) = \{1,2\}$ and $out(\cdot \texttt{ :: } \cdot \texttt{ ==> } \cdot) = \{3\}$. 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 those

[2] It subsumes context unification whose decidability is a long-standing open problem [11].

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.

Before defining well-modedness, we introduce the notation $vars(E)$ for a set of variables occurring in expression E, and $vars(E,\{p_1,...,p_n\}) = \cup_{i=1}^{n} vars(E|_{p_i})$, where $E|_{p_i}$ is the standard notation for a subexpression of E at position p_i. The symbol $\mathscr{V}_{\mathtt{an}}$ stands for the set of anonymous variables. A ground expression contains no variables. Then well-modedness of queries and clauses are defined as follows:

Definition 1 *A query* $\mathtt{L_1},\ldots,\mathtt{L_n}$ *is well-moded iff it satisfies the following conditions for each* $1 \le i \le n$*:*

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

A clause $\mathtt{L_0}$ `:-` $\mathtt{L_1},\ldots,\mathtt{L_n}$ *is well-moded, iff the following conditions are satisfied for each* $1 \le i \le n$*:*

- $vars(\mathtt{L_i}, in(\mathtt{L_i})) \cup vars(\mathtt{L_0}, out(\mathtt{L_0})) \subseteq \cup_{j=0}^{i-1} vars(\mathtt{L_j}, out(\mathtt{L_j})) \setminus \mathscr{V}_{\mathtt{an}}$.
- *If* $\mathtt{L_i}$ *is a negative literal, then* $vars(\mathtt{L_i}, out(\mathtt{L_i})) \subseteq \cup_{j=1}^{i-1} vars(\mathtt{L_j}, out(\mathtt{L_j})) \cup \mathscr{V}_{\mathtt{an}} \cup vars(\mathtt{L_0}, in(\mathtt{L_0}))$.
- *If* $\mathtt{L_0}$ *and* $\mathtt{L_i}$ *are* ρ*Log literals with the strategy terms* $\mathtt{st_0}$ *and* $\mathtt{st_i}$*, respectively, then* $vars(\mathtt{st_i}) \subseteq vars(\mathtt{st_0})$.

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. For well-moded programs and queries, PρLog uses Prolog's depth-first inference mechanism with the 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 :: h1 ==> h2`, then PρLog finds a (renamed copy of a) program clause `st' :: h1' ==> h2' :- body` such that $\mathtt{st}'$ matches `st` and $\mathtt{h1}'$ matches `h1` with a substitution σ. Then, it replaces the selected literal in the query with the conjunction of $\mathtt{body}\sigma$ and a literal that forces matching `h2` to $\mathtt{h2}'\sigma$, applies σ to the rest of the query and continues. Success and failure are defined in the standard way. Backtracking allows to explore other alternatives that may come from matching the (input positions in the) selected query literal to the (input positions in the) head of the same program clause in a different way, or to the (input positions in the) head of another program clause. Negative ρ-literals are processed by the standard negation-as-failure rule.

3 Strategic Programming

Strategies can be combined to express in a compact way many tedious small step transformations. These combinations give more control on transformations. PρLog provides a library of several predefined strategy combinators. Most of them are standard. The user can write her own strategies in PρLog or extend the Prolog code of the library. Some of the predefined strategies and their intuitive meanings are the following:

- `id` transforms a hedge in itself and never fails.
- $\mathtt{compose}(\mathtt{st_1},\mathtt{st_2},\ldots,\mathtt{st_n})$, $n \ge 2$, first transforms the input hedge by $\mathtt{st_1}$ and then transforms the result by $\mathtt{compose}(\mathtt{st_2},\ldots,\mathtt{st_n})$ (or by $\mathtt{st_2}$, if $n = 2$). Via backtracking all possible results can be obtained. Fails if either $\mathtt{st_1}$ or $\mathtt{compose}(\mathtt{st_2},\ldots,\mathtt{st_n})$ fails.
- $\mathtt{choice}(\mathtt{st_1},\ldots,\mathtt{st_n})$, $n \ge 1$, returns a result of a successful application of some $\mathtt{st_i}$ to the input hedge. It fails if all $\mathtt{st_i}$'s fail. By backtracking it can return all outputs of all $\mathtt{st_i}$ applications.

- `first_one`$(\mathtt{st}_1,\ldots,\mathtt{st}_n)$, $n \geq 1$, selects the first $\mathtt{st}_i$ that does not fail and returns only one result of its application to the input hedge. It fails if all $\mathtt{st}_i$'s fail. It's variation, `first_all`, returns via backtracking all the results of the application of $\mathtt{st}_i$ to the input hedge.
- `nf`(`st`) computes a normal form of the input hedge with respect to `st`. Never fails. Backtracking returns all normal forms.
- `iterate`(`st`,N) starts transforming the input hedge with `st` and returns a result (via backtracking all the results) obtained after N iterations.
- `map1`(`st`) maps `st` to each term in the input hedge and returns the result hedge. Backtracking generates all possible output hedges. `st` should operate on a single term and not on an arbitrary hedge. Fails if `st` fails on at least one term from the input hedge. `map` is a variation where the single-term restriction is removed.
- `interactive` takes a strategy from the user, transforms the input hedge by it and waits for the further user instruction (another strategy to be applied to the result hedge or to finish).
- `rewrite`(`st`) applies to a single term (not to an arbitrary hedge) and rewrites it by `st` (which also applies to a single term). Via backtracking it is possible to obtain all the rewrites. The input term is traversed in the leftmost-outermost manner. Note that `rewrite`(`st`) can be easily implemented inside PρLog:
```
rewrite(i_str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
    i_str :: i_Redex ==> i_Constractum.
```

Due to lack of space, we can not bring here examples that demonstrate these strategies. The users can define own strategies in a program either by writing clauses for them or using abbreviations like, e.g., `prove := nf(first_one(success, inference_step, failure))` for a simple (propositional) proof procedure, which abbreviates the clause (that can work on hedges of sequents):

```
prove :: s_X ==> s_Y :-
   nf(first_one(success, inference_step, failure)) :: s_X ==> s_Y.
```

Of course, the strategies `success`, `inference_step`, `failure` etc., operating on hedges of sequents, have to be provided.

4 Implementing Rewriting Strategies

Leftmost-Outermost and Outermost Rewriting. As mentioned above, the `rewrite` strategy traverses a term in leftmost outermost order to rewrite subterms. For instance, if the strategy `strat` is defined by the rules `strat :: f(i_X) ==> g(i_X)` and `strat :: f(f(i_X)) ==> i_X`, then for the goal `rewrite(strat) :: h(f(f(a)),f(a)) ==> i_X` we get, via backtracking, four instantiations for the variable `i_X`, in this order: `h(g(f(a)),f(a))`, `h(a,f(a))`, `h(f(g(a)),f(a))`, and `h(f(f(a)),g(a))`. If we want to obtain *only one result*, then it is enough to add the cut predicate in the goal: `rewrite(strat) :: h(f(f(a)),f(a)) ==> i_X,!` returns only `h(g(f(a)),f(a))`. On the other hand, if we want to get *all the results of leftmost-outermost rewriting,* we have to find the first redex and rewrite it in all possible ways (via backtracking), ignoring all the other redexes:

```
rewrite_left_out(i_str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
     i_str :: i_Redex ==> i_, !,
     i_str :: i_Redex ==> i_Contractum.
```

The goal `rewrite_left_out(strat) :: h(f(f(a)),f(a)) ==> i_X` gives two instantiations for `i_X`: `h(g(f(a)),f(a))` and `h(a,f(a))`.

To return *all the results of outermost rewriting* we find an outermost redex and rewrite it. Backtracking returns all the results for all outermost redexes.

```
rewrite_out(i_str) :: i_X ==> i_Y :-
     i_str :: i_X ==> i_, !,
     i_str :: i_X ==> i_Y.

rewrite_out(i_str) :: f_F(s_1,i_X,s_2) ==>f_F(s_1,i_Y,s_2) :-
     rewrite_out(i_str) :: i_X ==> i_Y.
```

The goal `rewrite_out(strat) :: h(f(f(a)),f(a)) ==> i_X` gives three answers, in this order: `h(g(f(a)),f(a))`, `h(a,f(a))`, and `h(f(f(a)),g(a))`.

Leftmost-Innermost and Innermost Rewriting. Implementation of innermost strategy in PρLog is slightly more involved than the implementation of outermost rewriting. It is not surprising since the outermost strategy takes an advantage of the PρLog built-in term traversal strategy. We could have modified the PρLog source by simply changing the order of two rules in the matching algorithm to give preference to the rule that descends deep in the term structure. It would change the term traversal strategy from leftmost-outermost to leftmost-innermost. Another way would be to build term traversal strategies into PρLog (like it is done in ELAN and Stratego, for instance) that would give the user a possibility to specify the needed traversal inside a PρLog program. However, here our aim is demonstrate that rewriting strategies can be implemented quite easily inside PρLog. For the outermost strategy it has already been shown. As for the innermost rewriting, if we want to obtain *only one result by leftmost-innermost strategy*, we first check whether any argument of the selected subterm rewrites. If not, we try to rewrite the subterm and if we succeed, we cut the alternatives. The way how matching is done guarantees that the leftmost possible redex is taken:

```
rewrite_left_in_one(i_str) :: c_Ctx(f_F(s_Args)) ==> c_Ctx(i_Contractum) :-
     rewrites_at_least_one(i_str) :: s_Args =\=> i_,
     i_str :: f_F(s_Args) ==> i_Contractum, !.

rewrites_at_least_one(i_str) :: (s_,i_X,s_) ==> true :-
     rewrite(i_str) :: i_X ==> i_, !.
```

To get *all results of leftmost-innermost rewriting*, we check whether the selected subterm is an innermost redex. If yes, the other redexes are cut off and the selected one is rewritten in all possible ways:

```
rewrite_left_in(i_str) :: c_Context(f_F(s_Args)) ==> c_Context(i_Contractum) :-
     rewrites_at_least_one(i_str) :: s_Args =\=> i_,
     i_str :: f_F(s_Args) ==> i_, !,
     i_str :: f_F(s_Args) ==> i_Contractum.
```

If `strat` is the strategy defined in the previous section, then we have only one answer for the goal `rewrite_left_in(strat) :: h(f(f(a)),f(a)) ==> i_X`: the term `h(f(g(a)),f(a))`. The same term is returned by `rewrite_left_in_one`.

Finally, `rewrite_in` computes *all results of innermost rewriting* via backtracking:

```
rewrite_in(i_str) :: f_F(s_Args) ==> i_Y :-
     rewrites_at_least_one(i_str) :: s_Args =\=> i_,
     i_str :: f_F(s_Args) ==> i_Y.

rewrite_in(i_str) :: f_F(s_1,i_X,s_2) ==> f_F(s_1,i_Y,s_2) :-
     rewrite_in(i_str) :: i_X ==> i_Y.
```

The goal `rewrite_in(strat) :: h(f(f(a)),f(a)) ==> i_X` returns two instantiations of `i_X`: `h(f(g(a)),f(a))` and `h(f(f(a)),g(a))`.

5 Concluding Remarks

We gave a brief overview on strategies in PρLog and showed how rewriting strategies can be compactly and declaratively implemented. PρLog extends Prolog with transformation rules over hedges, controlled by strategies, and is available for downloading from

`http://www.risc.uni-linz.ac.at/people/tkutsia/software.html`.

References

[1] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. *ENTCS*, 4, 1996.

[2] Y. Caseau, F.-X. Josset, and F. Laburthe. Claire: combining sets, search and rules to better express algorithms. *Theory Pract. Log. Program.*, 2(6):769–805, 2002.

[3] H. Cirstea and C. Kirchner. The rewriting calculus - Part I and II. *Logic Journal of the IGPL*, 9(3), 2001.

[4] 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.

[5] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *2nd IEEE Symposium on Logic Programming*, pages 29–38, 1985.

[6] T. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program*, 37(1–3):95–138, 1998.

[7] T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *LPAR05*, volume 3835 of *LNAI*, pages 215–229. Springer, 2005.

[8] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[9] M. Marin and T. Kutsia. Foundations of the rule-based system ρlog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

[10] N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.

[11] RTA List of Open Problems. Problem #90. Are context unification and linear second order unification decidable? http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html, 2009.

[12] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. In *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[13] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA'01*, volume 256 of *LNCS*. Springer, 2001.