Reports of Enlarged Session of the
Seminar of I. Vekua Institute
of Applied Mathematics
Volume 24, 2010

INFERENCE MECHANISM OF PρLog

Dundua B.

Abstract. We describe the inference mechanism of the PρLog language: an extension of logic programming with advanced rule-based programming features for hedge transformations, strategies, and regular constraints.

Keywords and phrases: Rule based programming, logic programming, context sequence matching, rewriting.

AMS subject classification: 68N17, 03B70, 68T15, 68Q42.

PρLog [1] (pronounced Pē-rō-log) is a Prolog implementation of the ρLog calculus [2], which extends the host language with strategic conditional transformation rules. These rules (basic strategies) define transformation steps on finite, possible empty, sequences consisting of terms or sequence variables. Such sequences are called hedges. 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 uses four different kinds of variables in one framework: individual, sequence, function, and context variables. It allows to traverse hedges in single/arbitrary width (with individual and sequence variables) and terms in single/arbitrary depth (with functional and context variables). These features facilitate flexibility in matching, providing a possibility to extract an arbitrary subhedge from a hedge, or to extract subterms at arbitrary depth.

More formally, terms and hedges in PρLog are built over unranked function symbols and the already mentioned four kinds of variables. These sets are disjoint. Here 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 formally defined by the grammars:

$$\mathrm{t} ::= \mathtt{i\_X} \mid \mathrm{f(h)} \mid \mathtt{f\_X}\mathrm{(h)} \mid \mathtt{c\_X}\mathrm{(t)}$$

$$\mathrm{h} ::= \mathrm{t} \mid \mathtt{s\_X} \mid \mathrm{eps} \mid (\mathtt{h\_1}, \mathtt{h\_2})$$

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 $\sigma =$`{c_Ctx↦f(hole), i_Term↦g(s_X),f_Funct↦g,s_Hedge1↦eps, s_Hedge2↦(b,c)}` and 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↦(),i_X↦a,s_2↦(f(b),c)}` and other one by the matcher `{s_1↦f(a),i_X↦b, s_2↦c}`. Similarly, the term `c_X(f_Y(a))` matches the term `f(a,g(a))` with the matchers `{c_X↦f(hole,g(a)),f_Y↦f}` and `{c_X↦f(a,g(hole)),f_Y↦g}`. An algorithm to solve matching problems in the described language has been introduced in [3].

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:

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

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.

We add regular constraints to matching problems to restrict the set of computed matchers. For instance, matching `c_X(f_Y(a))` to `f(a,g(a))` under the constraint `c_X in f(a,g(hole)*)`[1] gives one matcher `{c_X↦f(a,g(hole)),f_Y↦g}` instead of two for the unconstrained case mentioned earlier.

A ρLog *atom* (ρ-atom) is a quadruple consisting of a term `st` (a *strategy*), two 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 Prolog

[1] 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.

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 occurrences 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 [4] adapted to ρ-clauses. In these rules below, P stands for a program and Q denotes a query. `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**

$$\texttt{st :: h1 ==> h2 where R} \wedge Q \rightsquigarrow$$
$$\sigma(\texttt{body} \wedge (\texttt{id :: h2' ==> h2 where R}) \wedge Q)$$

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**

$$\texttt{id :: h1 ==> h2 where R} \wedge Q \rightsquigarrow \sigma(Q)$$

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

NF: **Negation as Failure**

$$(\texttt{st :: h1 =\textbackslash=> h2 where R}) \wedge Q \rightsquigarrow Q$$

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

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 [2] and which basically is based on the same notion for normal logic programs [5]. 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), 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.

Strategies control rule applications. They can be either user-defined or built-in, ground or contain variables, 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), `nf(st)` (computes a normal form of the input hedge with respect to st, if an application of `st` to a hedge fails, then `nf(st)` returns that hedge itself), etc.

We give below an example that illustrates inference mechanism of PρLog.

The following PρLog program rewrites term with respect to strategy `st` (the first clause implements rewriting a term by some rule `i_Str` and the second one gives one specific rule).

```
rewrite(i_Str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
     i_Str :: i_Redex ==> i_Contractum.
st :: f(s_X) ==> g(s_X).
```

The query `rewrite(st) :: f(f(a),b)  ==> i_X` , by the rule R, using the first clause and a substitution {`i_Str` →`st`, `c_Context` →`hole`, `i_Redex`→`f(f(a),b)`} , gives a new query:

`st ::  f(f(a),b) ==> i_Contractum, id :: i_Contractum ==> i_X`.

From here, again by the rule R, using the second clause and a substitution {`S_X`→`(f(a),b)`}, we obtain the query:

`id ::  g(f(a),b) ==> i_Contractum, id :: i_Contractum ==> i_X`.

The first subgoal is succeeds, applying to it the rule Id and the substitution {`i_Contractum`→`g(f(a),b)`}. The remaining query `id :: g(f(a),b) ==> i_X` can be satisfied by the same rule, using the substitution {`i_X`→`g(f(a),b)`}. Hence, the derivation is successful and PρLog returns the instantiation of the variables from the original query `i_X=g(f(a),b)`.

Meaning: The term `f(f(a),b)` can be rewritten by the rule `st` into `g(f(a),b)`.

If one wants to compute more answers, backtracking is initiated, which forces the query `rewrite(st) :: f(f(a),b)  ==> i_X` to be resolved against the first clause by the rule R and with a different substitution {`i_Str`→`st`, `c_Context` →`f(hole,b)`, `i_Redex`→`f(a)`} it gives a new query:

`st ::  f(a) ==> i_Contractum, id :: i_Contractum==> i_X`.

and so on.

For a more detailed presentation of the features and applications of PρLog we refer to [6,7].

Acknowledgments. This research has been funded by the Georgian National Science Foundation (ref. YS09 2 1-120 and 09 184 1-120).

REFERENCES

1. Dundua B., Kutsia T. PρLog. http://www.risc.uni-linz.ac.at/people/tkutsia/software.html.

2. Marin M., Kutsia T. Foundations of the rule-based system rholog. *J. Appl. Non-Classical Logics*, **16**, 1-2 (2006), 151-168.

3. Kutsia T., Marin M. Matching with regular constraints. In Voronkov A., Sutcliffe G., editor, *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 3835 of *LNAI*, Springer, (2005), 215-229.

4. Apt K., Bol R. Logic programming and negation: A survey. *J. Log. Programm.*, **19** (1994), 9-71.

5. Deransart O., Maluszynski J. Relating logic programs and attribute grammars. *J. Log. Programm.*, **2**, 2 (1985), 119–155.

6. Coelho J., Dundua B., Florido M., Kutsia T. A Rule-based Approach to xml Processing and Web Reasoning. *In P. Hitzler and T. Lukasiewicz, editors, RR, volume 6333 of Lecture Notes in Computer Science*, pages 164-172. Springer, 2010.

7. Dundua B., Kutsia T., Marin M. Strategies in PρLog. EPTCS, 15:32-43, 2010.

Received 3.08.2010; revised 13.10.2010; accepted 15.11.2010.

Author's address:

Dundua B.
I. Vekua Institute of Applied Mathematics of
Iv. Javakhishvili Tbilisi State University
2, University St., Tbilisi 0186
Georgia
E-mail: bdundua@gmail.com