PρLog: Logic Programming, Rules, and Strategies

Besik Dundua
Kutaisi International University, and
I.Vekua Institute of Applied Mathematics,
I.Javakhishvili Tbilisi State University
besik.dundua@kiu.edu.ge

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 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 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. Following Prolog, PρLog is also untyped, but values of sequence and context variables can be constrained by regular hedge or tree languages.

In this poster we explain by simple examples how PρLog system works. The sources 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 version 7.2.3 or later.

Programming in PρLog

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

```
merge_doubles :: (s_X, i_X, s_Y, i_X, s_Z) ==> (s_X, i_X, s_Y, s_Z).
```

The code, as one can see, is pretty short. `merge_doubles` is the strategy name. It says that if the sequence in `(s_X, i_X, s_Y, i_X, s_Z)` contains duplicates (expressed by two copies of the individual variable `i_X`, which can match individual terms) somewhere, then from these two copies only the first one should be kept in `(s_X, i_X, s_Y, s_Z)`. 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.

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) ==> s_Result.
```

First, PρLog returns the result `s_Result = (1, 2, 3, 2)`. Like in Prolog, the user may ask for more solutions, and, via backtracking, PρLog gives the second answer `s_Result = (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

```
merge_all_doubles :: s_X ==> s_Y :- nf(merge_doubles) :: s_X ==> 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) ==> s_Result.
```

we get a single answer `s_Result = (1, 2, 3)`.

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

```
merge_all_doubles :: s_X ==> s_Y :-
    first_one(nf(merge_doubles)) :: s_X ==> 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 `hole`. 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)) ==>c_Context(f_Fun(s_X)) :-
    merge_doubles :: s_X ==> 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 (individual, sequence, function, context), 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))) ==> i_Result.
```

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

```
{i_Result = f(g(a, b, h(c, c)), h(c), g(a, a, b, h(c)))},
{i_Result = f(g(a, b, a, h(c)), h(c), g(a, a, b, h(c)))},
{i_Result = 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(hole, 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))) ==> c_C(h(c,c)).
```

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

```
{c_C = f(g(a, b, hole), h(c), g(a, a, b, h(c)))},
{c_C = f(g(a, b, a, hole), 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)))==> s_Result.
```

we get a single answer `{s_Result = f(g(a, b, h(c)), h(c))}`.

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

Acknowledgements

This work was supported by Shota Rustaveli National Science Foundation of Georgia under the project FR-21-7973.