Proceedings of I. Vekua Institute of Applied Mathematics
Vol. 58, 2008

PROGRAMMING WITH SEQUENCE AND CONTEXT VARIABLES

Dundua B.

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
A-4040 Linz, Austria
e.mail: dundua@risc.uni-linz.ac.at

Abstract. Context and sequence variables make matching flexible and expressive. In pattern matching based programming, they enhance capabilities of the language to write compact, declarative, and readable code. PρLog is a tool that extends Prolog with context sequence matching and strategic conditional transformation rules. In this paper we briefly describe PρLog, concentrating on the usage of context and sequence variables in programming.

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

AMS subject classification (2000): 68N17; 03B70; 68T15; 68Q42.

1. Introduction

In recent years usefulness of sequence and context variables has been shown in various areas of mathematics and computer science. Sequence variables are placeholders for arbitrarily long finite sequences of expressions and have applications in programming [16], XML querying and transformation [4], term rewriting [7], knowledge engineering and artificial intelligence [15], automated reasoning [13, 6]. Context variables are placeholders for contexts, which are functional expressions whose applicative behavior is to replace a special constant (the constant `hole`) with the expression given as argument. They have applications in compositional semantics of natural language [11, 8]. Combination of this variables in a single framework allows flexible term traversal in arbitrary width (with sequence variables) and in arbitrary depth (with context variables). In addition, we can restrict possible values of sequence and context variables by constraining sequence variables by regular hedge expressions and context variables by regular tree expressions.

Solving equations between hedges containing context and sequence variables is a challenging task in unification theory, since decidability of context unification is still an open problem [14]. In [9] matching between hedges over context and sequence variables (context sequence matching) has been studied and a matching algorithm for both unconstrained matching and for matching with regular constraints has been given. Moreover, this matching algorithm is finitary and always computes minimal complete set of matchers.

PρLog is a system for rule-based programming [5] based on the calculus described in [10]. It integrates powerful pattern matching mechanisms with sequence and context variables and regular constraints in a single framework. These capabilities together with strategies enable highly declarative programming style that is expressive enough to support concise implementations for: specifying and prototyping deductive systems, solvers for various equational theories, tools for XML querying and transformation, etc. We do not elaborate on the role of strategies in PρLog system here, but, rather, focus on demonstrating the expressive power of context and sequence variables in PρLog programs.

PρLog was inspired by rule-based programming languages such as ELAN [1] and Maude [3], but the computational mechanism is different. ELAN is based on the ρ-calculus, Maude is based on the rewriting logic [2, 3], Whereas PρLog is based on the principles of logic programming with negation as finite failure [10] and extends logic programming with sequence, context, functional variables, regular expressions and strategic conditional transformation rules for hedges.

2. Matching Power

Terms `t` and hedges `h` are main syntactic categories of PρLog language and are constructed in a standard way:

$$\begin{array}{ll} \mathtt{t} ::= \mathtt{hole} \mid \mathtt{i\_X} \mid \mathtt{f(h)} \mid \mathtt{f\_X(h)} \mid \mathtt{c\_X(t)} & \text{terms} \\ \mathtt{h} ::= \mathtt{eps} \mid \mathtt{t} \mid \mathtt{s\_X} \mid (\mathtt{h}, \mathtt{h}) & \text{hedge} \end{array}$$

where `i_X`, `f_X`, `c_X`, `s_X` are from countable sets of individual, functional, context and sequence variables respectively, `f` ranges over a countably set of functional symbols, `hole` is a special function symbol called the hole symbol, `eps` stands for the empty hedge and is omitted whenever it appears as a subhedge of another hedge. A *context* is a term with a single occurrence of the `hole` constant. Application of a context to a term `t` is a term derived by replacing the hole in the context with `t`. We write anonymous individual variables as `i_`, sequence variables as `s_` , context variables as `c_` and functional variables as `f_`.

A *substitution* is a mapping from individual variables to hole-free terms that are not sequence variables, 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 and function variables are mapped to themselves, all but finitely many sequence variables are mapped to themselves considered as singleton sequences, and all but finitely many context variables are mapped to themselves applied to the `hole`. A substitution is a `solution` of the matching problem `t1 << t2`, where t2 is a ground term, if `t1 solution=t2`

Context sequence matching is the main computational mechanism in the PρLog system. In this paper we describe neither the algorithm nor its implementation (context sequence matching is in general not unique and hence the system has to choose matcher). Instead, we just demonstrate the algorithm on simple examples. First we consider an example without regular constraints.

Example 1. Context sequence matching without regular constraints

```
match([c_C(h(s_X,b,s_Y))<< f(a,b,h(a,b,b,h(a,b)))], Solution).
Solution = [c_C ---> f(a, b, hole), s_X ---> a,
           s_Y ---> (b, h(a, b))] ;
Solution = [c_C ---> f(a, b, hole), s_X ---> (a, b),
           s_Y ---> h(a, b)] ;
Solution = [c_C ---> f(a, b, h(a, b, b, hole)), s_X ---> a,
           s_Y ---> eps] ;
false.
```

If we restrict the possible values of the sequence variable `s_X` by the regular hedge expression `sstar(a)` (the language generated by it is {`eps, a, (a,a), (a,a,a),...`}) and the possible values of the context variable `c_C` by the regular tree expression `f(s_,hole)` (the language generated by it is {`f(s_,hole)`}), then we have:

```
match([c_C(h(s_X,b,s_Y))<< f(a,b,h(a,b,b,h(a,b))),
      [s_X, sstar(a)],[c_C, f(s_,hole)]],Solution).
Solution = [c_C ---> f(a, b, hole), s_X ---> a,
s_Y ---> (b, h(a, b))] ;
false.
```

3. Programming

A PρLog program is a collection of Prolog clauses and clauses in the form `st :: h1 ==> h2 where C :- body` where `st` stands for strategies, `h1, h2` for hedges and `C` for regular constraints to restrict sequence and context variables occurring in `h1` and `h2`. `body` is conjunction of prolog literals and PρLog atoms in the form `st :: h1 ==> h2 where C` or its negation. We require PρLog clauses and those prolog clauses that define a predicate that occurs in the body of some PρLog clause to be well-moded [12, 10]. A PρLog query is a conjunction of PρLog and Prolog literals satisfying well-modedness property. We require the restriction of well-modedness to guarantee that each execution step is performed using matching [9] and not unification (whose decidability is not known)[14].

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 PρLog atom of the form `st :: h1 ==> h2 where C`, then PρLog finds a (renamed copy of a) program clause `st' :: h1' ==> h2' where C' :- body` and computes matcher σ `= [st'<<st,h1'<<h1,C']`. Then, it replaces

the selected literal in the query with the conjunction of `body`σ and a literal `id :: h2'`σ `==> h2 where C`σ, 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. If selected literal is negation of the atom `st :: h1 ==> h2 where C`, then it is processed by the standard negation-as-failure rule.

Application of a PρLog program clauses (we sometime call it rule) in the form `st :: h1 ==> h2 where C :- body` to a query may return several results, which may come from multiple matchers. In order to take into account non-determinism and set of results, and to control rule application, the concept of strategy is introduced. PρLog provides several predefined strategy operators, such as `compose` (sequential composition), `choice` (nondeterministic choice), `nf` (normalization), `first_one` (leftmost applicable strategy), `rewrite` (term rewriting extended to hedges), `map1` (map of the application of a strategy to all terms of the input hedge), etc. for building strategies which specifies application of sequence of rules to a given input.

To demonstrate expressive power of sequence and context variables and role of strategies in programming, we show how some problems can be implemented in PρLog.

Example 2. The following program illustrates how bubble sort can be implemented in PρLog.

```
swap(f_Ordering) :: (s_X, i_I, s_Y, i_J, s_Z) ==>
                         (s_X, i_J, s_Y, i_I, s_Z)  :-
                              not( f_Ordering(i_I,i_J)).

bubble_sort(f_Ordering) := first_one(nf(swap(f_Ordering))).
```

This algorithm takes two elements from a given sequence and compares them with respect to `f_Ordering`. If the elements are not determined to be ordered by `f_Ordering`, then they are swapped. `nf` applies `swap` repeatedly until impossible, which leads to a sorted sequence.

Note that,

`bubble_sort(f_Ordering) := first_one(nf(swap(f_Ordering)))` is an abbreviation of the clause

```
        bubble_sort(f_Ordering) :: s_X ==> s_Y :-
    first_one(nf(swap(f_Ordering))) :: s_X ==> s_Y.
```

The PρLog query

```
?(bubble_sort(=<)::(1,3,4,3,2) ==> s_X,Result).
```

produces the result

```
Result = [s_X ---> (1, 2, 3, 3, 4)] ;
false.
```

The following example illustrates how easily one-step rewriting can be implemented in PρLog.

Example 3. One-step rewriting without regular constraint.

```
rewrite_one_step(i_Str) :: c_X(i_X) ==> c_X(i_Y) :-
                  i_Str :: i_X ==> i_Y.
```

If we want to rewrite direct subterms of the function symbol f, then we have to restrict value of the context variable `c_X` by the regular tree expressions `c_(f(s_,hole,s_)` as is shown in Example

Example 4. One-step rewriting with regular constraint.

```
rewrite_under_f(i_Str) :: c_X(i_X) ==>
        c_X(i_Y) where ([c_X in c_(f(s_,hole,s_))]) :-
                    i_Str :: i_X ==> i_Y.
```

Now, we can see how PρLog rewrites terms using rewriting defined by Example and Example with respect to one-rule rewriting system $a \to b$.

```
?(rewrite_one_step(st):: f(f(g(a),a),a) ==> s_x,Result).
Result = [s_x ---> f(f(g(b), a), a)] ;
Result = [s_x ---> f(f(g(a), b), a)] ;
Result = [s_x ---> f(f(g(a), a), b)] ;
false.

?(rewrite_under_f(st):: f(f(g(a),a),a) ==> s_x,Result).
Result = [s_x ---> f(f(g(a), a), b)] ;
Result = [s_x ---> f(f(g(a), b), a)] ;
false.
```

Leftmost innermost strategy traversals given term by leftmost innermost order and returns first successful result nondeterministically. Implementation of leftmost innermost strategy is given in Example

Example 5. Leftmost innermost strategy

```
rewrite_left_in(i_str) :: c_Ctx(f_F(s_Args)) ==>
                          c_Ctx(i_Contractum) :-
      rewrites_at_least(i_str) :: s_Args ==> false,
      i_str :: f_F(s_Args) ==> i_Contractum, !.
rewrites_at_least(i_str) :: (s_,i_X,s_) ==> true :-
     rewrite(i_str) :: i_X ==> i_, !.
rewrites_at_least(i_str) :: s_ ==> false.
```

Application of the leftmost innermost rewriting strategy to a term `f(f(g(a),a),a)` results

```
?(rewrite_left_in(st):: f(f(g(a),a),a) ==> s_X,Result).
Result = [s_X ---> f(f(g(b), a), a)] ;
false.
```

4. Future work

We plan to further experiment with PρLog, implementing various proving systems, solvers and simplifiers on it. We also would like to extend PρLog calculus by second order terms containing sequence variables and to implement it in the new version of PρLog system.

REFERENCES

1. Baldan P., Bertolissi C., Cirstea H., Kirchner C. A rewriting calculus for cyclic higher-order term graphs. *Mathematical. Structures in Comp. Sci.*, **17**, 3 (2007), 363–406.

2. Borovanský P., Kirchner C., Kirchner H., Moreau P.E. Elan from a rewriting logic point of view. *Theor. Comput. Sci.*, **285**, 2 (2002), 155–185.

3. Clavel M., Durán F., Eker S, Lincoln P., Narciso M.O., Meseguer J., Quesada J. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

4. Coelho J., Florido M. Clp(flex): Constraint logic programming applied to xml processing. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. Proc. of Confederated Int. Conferences, volume 3291 of LNCS*, pages 1098–1112. Springer, 2004.

5. Dundua B., Kutsia T., Marin M. Pρlog. http://www.risc.uni-linz.ac.at/people /tkutsia/software.html.

6. Ginsberg M.L. The MVL theorem proving system. *SIGART Bull.*, **2**, 3 (1991), 57–60.

7. Hamana M. Term rewriting with sequences. Technical Report 97-20, RISC, Johannes Kepler University, Linz, Austria, 1997.

8. Koller A. Evaluating context unification for semantic underspecification. In *Proceedings of the Third ESSLLI Student Session*, Saarbrücken, Germany, 1998.

9. 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*, pages 215–229. Springer, 2005.

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

11. Niehren J., Pinkal M. A uniform approach to underspecification and parallelism. In *In Proceedings ACL'97*, pages 410–417, 1997.

12. Ohlebusch E. Termination of logic programs: Transformational methods revisited. *Appl. Algebra Eng. Commun. Comput.*, **12**, 1/2 (2001), 73–116.

13. Paulson L. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

14. RTA List of Open Problems. Problem #90. Are context unification and linear second order unification decidable? http://rtaloop.mancoosi.univ-paris-diderot.fr/

15. Volpano D.M. Haskell-style overloading is np-hard. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 88–94, 1994.

16. Wolfram S. *The Mathematica Book*. Wolfram Media, fifth edition, August 2003.

Received: 5.02.2008; revised: 2.07.2008; accepted: 16.10.2008.