AMALGAMATING CCP WITH PROLOG

Tudor Muresan
Rodica Potolea
Smaranda Muresan
Technical University of Cluj-Napoca
{tmuresan,potolea}@cs.utcluj.ro smara@maria.utcluj.ro

Abstract The CCP language accepts to amalgamate Prolog clauses with guarded clauses of cp(↓,|,&) family languages. The CCP language has sequential and parallel OR operators and a static deep terms annotation. A single guard operator is used, which together with OR operators, provides 'don't care' and 'don't know' nondeterministic behaviour. The allowed guards must be 'wait free'. The depth first search strategy with backtracking is used. Annotated terms synchronisation provides a reactive behaviour, while sequential OR synchronisation with backtracking provides multiple solution searching. Logical conjunction comutativity is accomplished for CCP predicates.

Keywords: Logic programming, concurrency, don't care and don't know nondeterminism, freeze, implementation.

1. Introduction

The paper presents a CCP language that accepts to amalgamate Prolog clauses with guarded clauses of cp(↓,|,&) Saraswat's family languages [10], [11]. The CCP language has sequential OR ';' and parallel OR '.' operators (as in Parlog [2]) and a static deep terms annotation (as in cp(↓,|,&)). A single guard operator '!' is used, which together with '.' and ';' operators, provides 'don't care' respectively 'don't know' nondeterministic behaviour. The guards with a local environment may be deep, but they have to be 'wait free' [11]. A single depth first search strategy with backtracking is used (the one of Prolog with freeze). Annotated terms synchronisation provides a reactive behaviour (e.g. shared variables as communication channels), while sequential OR synchronisation with backtracking provides for multiple solution searching. Logical conjunction comutativity is accomplished for CCP predicates. So, the CCP language, behaves like languages with breadth first (fair) search strategy and/or encapsulated search [3], [4], [6], [13].

The preserved unfair search strategy of Prolog is balanced by allowing the side effects with their advantages. In this case the soundness is left in the charge of users.

Section 2 is a brief review of SLD resolution properties [7] with soundness proof of Prolog with freeze ($\mathbf{P_f}$) programs. This explains the $\mathbf{P_f}$ selection as target language for CCP program's translation. Section 3 presents CCP language as following: syntax language definition (3.1), $\mathbf{P_f}$ semantics definition, based on a meta-interpreter of $\mathbf{P_f}$ programs to standard Prolog (3.2). The subsection 3.3 deals with the CCP language semantics definition, by its translating schemes in $\mathbf{P_f}$. The equivalence relation on which the translation is based and its soundness proof are both supplied. Finally, Section 4 discusses about implementation and experimental results and Section 5 contains our conclusions.

2. Basic theoretical results

A SLD refutation procedure for a logic program P and a goal G uses a computation rule and a search strategy (rule). The computation rule chooses a subgoal from the sequence of goals to perform the derivation step. A SLD derivation is (said to be) *fair* if it ensure any subgoal selection in a finite number of steps (the depth first search strategy of Prolog is unfair).

The soundness and completeness of a fair SLD refutation has been proved, that is the equivalence between the logical consequence $(P \models G)$ and SLD refutation $(P \vdash G)$.

$$(P \models G) \quad \leftrightarrow \quad (P \vdash G)$$

If the negation as failure is taken into consideration, a SLDNF computation rule is said to be *safe* if it selects only ground negative literals and it does not interrupt the coresponding SLDNF finite failure subtree building.

If comp(P) is the Clark completion of a program P, and the SLDNF rule of computation is safe, the soundness and completeness of SLDNF refutation hold true.

$$(comp(P) \models G) \quad \leftrightarrow \quad (P \vdash G)$$

Moreover, the soundness and completeness of a SLD refutation is independent of the chosen computation rule (e.g. the current subgoal selection).

For the rest of the paper we consider P_f the logic program obtained from P, by enclosing any subgoal G_i of a clause into a freeze(Var,G_i) predicate, where Var belongs to the set of G_i variable, Var $\in$ SetVar(G_i).

If

$H\text{:- } B_1,..,B_i ,..,B_n \in P,$

then

$H\text{:- } B_1,..,\text{freeze}(Var, B_i),..,B_n \in P_f .$

A subgoal freeze(Var, B_i) is not selected as long as Var is unbound. On it's selection the equivalence

$\text{freeze}(Var, B_i) \leftrightarrow B_i$

takes place.

A fair SLD refutation for a program P_f and a goal G is achieved if the empty clause may be derived in a finite number of steps. This means that all the subgoals freeze(Var, B_i) have been actually selected. Taking into consideration the independence of the choice of the computation rule and the logic equivalence between freeze(Var, B_i) and B_i in the moment of the selection, we have:

Lemma : $(P_f \vdash G) \rightarrow (P \vdash G)$
and
Corollary: $(P_f \vdash G) \rightarrow (P \models G)$
(Soundness of SLD refutation for programs with freeze.)

The soundness of P_f programs makes possible the use of Prolog with freeze language as target language in the translation of CCP languages with don't care nondeterminism, for which the incompleteness is an intrinsic feature.

Even if the Prolog strategy is an unfair one, freeze does not introduce new exceptions from the theoretical model (comparing with those of standard Prolog). Moreover, freeze may improve the Prolog programs behaviour, making safe the negative literals selection. However, the Prolog strategy makes incompatible the use of freeze together with the cut ('!'), without imposing special restrictions (e.g. the wait free guards proposed by Saraswat [11]).

3. Syntax of CCP language

As primitives of syntax program specification we assume the notions of variables, terms and atoms. Moreover we consider annotated terms and annotated atoms.

An annotated term is a Prolog term that has the constants and variables post-annotated with "mode information" ('+', '-', '?'), as in Prolog modes specification. The terms may be deep annotated as in Saraswat's cp(↓,|,&) language [10]. An annotated atom is an atom, which has annotated terms as arguments.

Our CCP language provides the '!' commitment operation and sequential-OR ';' delimiter. So, one may amalgamate the standard Prolog clauses (without cut), assuming backtracking, with 'don't care commitment' alternatives. The syntax of the program is described below.

```
<program> ::= <sequence of clauses>
<sequence of clauses> ::= <clause> | <sequence of clauses>.<clause> |
                          <sequence of clauses>;<clause>
<clause> ::= <head>:-<body> | <head>:-<guard> ! <body>
<guard>,<body> ::= <sequence of atoms>
<sequence of atoms> ::= <atom> | <sequence of atoms>,<atom>
<head> ::= <atom> | <annotated atom>
```

The CCP programs have concurrent behavior due to annotated terms synchronization and the local environment of the guards.

3.2. Meta-interpreter of P_f

The operational semantics specification of freeze is done by defining a meta-interpreter of the logic language. Let solve(Goal) be a SLD refutation procedure for a logic program, using a depth first strategy:

```
solve(true).
solve((Goal,Rest_goal)):-
                  solve(Goal),
                  solve(Rest_goal).
solve(Goal):-
            clause(Goal,Body),
            solve(Body).
```

The logic programming language extended with `freeze(X,Goal)` predicates will have the SLD refutation procedure given by the meta-interpreter:

```
solve(Goal) :- solve_freeze(Goal,T\T,_).

solve_freeze(Goal) --> solve(Goal),solve_queue.

solve((Goal,Rest_Goal))-->solve_freeze(Goal),solve_freeze(Rest_Goal).
solve((Goal;Rest_Goal))-->solve_freeze(Goal);solve_freeze(Rest_Goal).
solve(true) -->.
solve(freeze(Var,Goal)) --> {nonvar(Var),!},solve_freeze(Goal).
solve(freeze(Var,Goal),Ti/(freeze(Var,Goal),To),Ti/To).
solve((Goal)) --> {clause(Goal,Body)},solve_freeze(Body).

solve_queue(T/T,T/T) :- var(T),!.
solve_queue((freeze(Var,Goal),Ti)/To,Qo):-
                                        var(Var),!, solve_queue(Ti/To,Q),
                                        resuspend(freeze(Var,Goal),Q,Qo).
solve_queue((freeze(_,Goal),Ti)/To,Qo):- solve_freeze(Goal,Ti/To,Qo).

resuspend(Freeze,Ti/To,(Freeze,Ti)/To).
```

The `solve_freeze(Goal,Qi,Qo)` predicate has two additional arguments, representing the queue of suspended goals before (`Qi`) and after (`Qo`) the goal (`Goal`) execution. The queue Q is represented as difference lists. The predicate `solve(freeze(Var,Goal),Qi,Qo)` execution adds a goal in the queue, if `Var` is unbound, otherwise it executes the corresponding goal. The `solve_queue/2` predicate execution takes from the queue the first suspended goal with its `Var` bound and executes it.

3.3. The CCP programs translation into P_f

The Prolog freeze(Var,G) predicate semantic as it is specified in the meta-interpreter in subsection 3.2, differs from that presented in Section 2, where fair computation rules are assumed. The (unfair) depth first strategy of Prolog has as computation rule the left-most unsuspended goal selection. This means that for unbound Var we have:

$$\text{freeze(Var,G)} \ \leftrightarrow^{b} \ \text{true},$$

the subgoal G being suspended. As a consequence, for Prolog depth first strategy with backtracking, regarding the conjunction we have the equivalence:

$$\text{freeze(Var,(G1,G2))} \ \leftrightarrow^{b} \ \text{freeze(Var,G1),freeze(Var,G2)} \qquad (1)$$

where $\text{Var} \in \text{SetVar(G1)} \cup \text{SetVar(G2)}$

On the other hand, for the disjunction we have:

$$\text{freeze(Var,(G1;G2))} \ \neg\leftrightarrow^{b} \ \text{freeze(Var,G1);freeze(Var,G2)} \qquad (2)$$

Although being logically equivalent, the left and right sides of relation (2) have different behaviour on backtracking. In order to have the same behaviour the use of the conjunction of freeze predicates must be imposed. For the disjunction we propose the following construction:

D = freeze(Var, (Flag=1,G1 ; off=A)) , freeze(Var, (Flag=2,G2 ; A=on))

where Flag, A $\notin$ SetVar(G1) $\cup$SetVar(G2).
Taking into consideration the subgoals conjunction equivalence, we have:

D $\leftrightarrow^{b}$ freeze(Var,((Flag=1,G1 ; off=A) , (Flag=2,G2 ; A=on)))

Or, by using the distributive rule, we obtain:

D $\leftrightarrow^{b}$ freeze(Var,(Flag=1,G1,A=on ; off=A,Flag=2,G2))

And because Flag,A $\notin$ SetVar(G1) $\cup$SetVar(G2) we have:

freeze(Var,(G1 ; G2)) $\leftrightarrow^{b}$ freeze(Var , (Flag=1,G1 ; off=A)),
freeze(Var , (Flag=2,G2 ; A=on)). (3)

That relation is used in the translation of CCP clauses into P_f. As a final remark, we mention that (1) and (3) relations express the equivalence of left and right sides as a behaviour on Prolog backtracking (together with the logic equivalence).

The translation is similar to that of from AKP [4] and Saraswat [10]. But here, we have a single guard operator and the nondeterminism is achieved by two OR operators: ‘;’ (don’t know) and ‘.’ (don’t care).

To each predicate call, a calling flag is attached. On the predicate call the flag is set (set_flag), tested at the beginning of each alternative clause of the predicate (active_flag) and reset on the success of the guard (reset_flag). In this way, the exclusive use of the ‘don’t know’ nondeterminism (‘;’ operator) provides to the guarded clauses the same operational semantics as the cut operator in Prolog. For the ‘don’t’ care nondeterminism (‘.’ operator) all the clauses are guarded clauses. Depending on the synchronisation asked by the predicate arguments annotation, many clauses may be concurrent (e.g. execution of the guards in their local environment). The first clause that executes the reset_flag predicate is selected, while the others clauses alternatives are forced to fail. On the execution of the guards there is no communication with the outer environments and the queues of the suspended goals at the beginning and at the end of the guard execution must be the same (wait free guards).

At meta-interpreter level, presented at Section 2, these facts are interpreted in the following way:

```
solve((Goal)),Qi,Qo):-  clause(Goal,(Guard!Body)),
                        solve_freeze(Guard,Qi,Qi),
                        solve_freeze(Body,Qi,Qo).
```

On the other hand on the direct execution in SICStus Prolog, the infix operator definition is added:

```
G ! B :- call_residue(G,Res) , ( Res=[] -> call(B) ; fail ).
```

The clauses ended on ‘;’ or ‘.’ may be amalgamated in any order with the above presented semantics. Bellow we present a CCP sequence in which all possible combinations appear, together with its P_f translation. For the time being, we do not consider the deep annotated arguments.

```
p(X+ , Y) :- g1(X,Y) ! b1(X,Y).
p(X+ , Y) :- b2(X,Y).
p(X+ , Y) :- g3(X,Y) ! b3(X,Y) ;
p(X+ , Y) :- b4(X,Y) ;
p(X+ , Y) :- g5(X,Y) ! b5(X,Y).
p(X+ , Y) :- b6(X,Y) ;
```

CCP sequence

```
p(X,Y) :- set_flag(Ip),
          freeze(X,(F=_,p(1,X,Y,Ip) ; off=V1)),
          freeze(X,(F=_,p(2,X,Y,Ip) ; V1=V2)),
          freeze(X,(F=_,p(3,X,Y,Ip) ; V2=V3)),
          freeze(V3,freeze(X,(F=_,p(4,X,Y,Ip) ; V3=V4)),
                    freeze(V4,freeze(X,(F=_,p(5,X,Y,Ip) ; V4=V5)),
                              freeze(X,(F=_,p(6,X,Y,Ip) ; V5=on))
                          )
               ).

p(i,X,Y,Ip):-active_flag(Ip),gi(X,Y)!reset_flag(Ip),bi(X,Y). i=1,3,5.
p(j,X,Y,Ip):-reset_flag(Ip),bj(X,Y).                          j=2,6.
p(k,X,Y,Ip):-active_flag(Ip),bk(X,Y).                         k=4.
```

Translation in P_f

For the p/2 CCP predicate (p/n in general) the P_f translated sequence has a clause p/2 (in general p/n), corresponding to the predicate call, and as many p/4 clauses (p/n+2 in general) as the number of clauses of the CCP predicate.

The p/2 clause contains a freeze conjunction, based on the equivalence (3). The sequential operator ‘;’ introduces suspension on the auxiliary variables (V3, V4) which become bound only on the failure of the precedent alternatives. The ‘.’ operator introduces only suspensions on annotated variables (X), making possible the concurrent execution of the alternatives (1, 2, 3 or 5, 6).

The p/4 clauses have two additional arguments: the first one, being the clause index(I), and the last one, being the calling flag (Ip).

The i-index form is generated for all clauses with explicit guard, that with j-index for the clauses without guard and don’t care nondeterminism, and the last one, with k-index, for the clauses without guard and don’t know nondeterminism.

The sequentiation obtained through suspension on additional variables (V3,V4) and the concurrent execution of the don’t care alternatives guards (with the above mentioned restrictions), make unnecessary the F=i subgoals in the equivalence relation (3).

Undeclared CCP clauses are assumed to be Prolog clauses and are not translated.

A real CCP program may contain clauses with deep annotated arguments, which means multiple variables suspension. This is ensured during the P_f translation by nested freeze containing unification of the patterns given by the annotated arguments.

For example, the clause bellow:

```
.
.
p([m(I+,R)|T]):-b(I,R)…                                   the i-th clause of p/1
.
.
```

is translated into:

```
p(A) :- set_flag(Ip),
      …
      freeze(A,(A=[B|_],
                   freeze(B,(B=m(C,_),
                             freeze(C,(C=_,p(i,A,Ip) ; Vi-1=Vi )
                                    )
                              )
                         )
                 )
            )
      …
```

Nested suspension on A, B, C variables can be noticed, along with annotated term pattern checking. The real translated sequence adds to each inner conjunction an OR alternative (; $V_{i-1}=V_i$) ensuring always the successful ending of the freeze predicate on resume. In this way on each call of p(A) the failure is produced only after trying all alternatives clauses.

If an initial goal has multiple solutions, some subgoals remain suspended on auxiliary variables (Var $\notin$ SetVar(G)), but on the execution with a findall (_,G,_) the end is achieved without suspended goals.

4. Implementation and experimental results

The implementations of the translator (subsection 3.3) and meta-interpreter (Section 2) for the CCP programs have been made by using SICStus Prolog V3.5. The source programs make possible the CCP clauses amalgamation with those of Prolog, using its built-in predicates, including those with side effects (except for the cut). In this way the operational semantics of Prolog is extended with 'data-flow' functionality, also present in guarded clause logic languages. In this way, all typical examples of CP [12], Parlog [2] and GHC might be run (e.g. producer/consumer interaction, object programming using delegation and examples with general reactive behaviour).

Representative for side effects is the problem of graph isomorphism, using the default logic technique. The main predicate verifies the vertex equivalence, by making pairs in the database using pair(Vertex1,Vertex2) predicate.

```
eq_vertex(V1+ ,V2+ ) :- pair(V1,V2) ! true ;
eq_vertex(V1+ , _  ) :- pair(V1, _) ! fail.
eq_vertex( _  ,V2+ ) :- pair( _,V2) ! fail ;
eq_vertex(V1+ ,V2+ ) :- assert(pair(V1,V2)) ;
eq_vertex(V1+ ,V2+ ) :- retract(pair(V1,V2)) ! fail ;
```

We observe that if V1 and V2 do not match (the failure of the first clause) and the database does not contradict the matching (don't care clauses 2 and 3), the pair(V1,V2) is asserted in the database (clause 4), which means the immediate hypothesis assumption. The 4th clause is the only one that allows backtracking (';' operator and lack of guard). In this way, on a later failure detection, clause 5 is executed, and it retracts the hypothesis (pair(V1,V2)), sending backward the failure.

The tested examples emphasise the subgoals comutativity in the CCP conjunction and the preservation of the side effects in the same order as the Prolog subgoals.

5. Conclusions

The paper proposes the CCP language, similar to cp(↓,|,&) family languages of Saraswat, language that extends the operational semantics of standard Prolog.

The model of translation from CCP to P_f, together with the soundness argumentation of P_f programs is original. The implementation is complete and operational. The depth first strategy search of Prolog and its backtracking mechanism are maintained. In this way the side effects capacity of Prolog is preserved.

The entire system together with the test programs are about to be installed on the web, as a part of developing the Grant CNCSU D3/111, 1997.

References

[1] F. Bueno, M. Jose Garcia de la Banda, M. Hermenegildo, U. Montanari and F. Rossi, From eventual to atomic and locally atomic CC programs: A concurrent semantics. In *Proceedings of the International Conference on Algebraic and Logic Programming*, 1994.

[2] S. Gregory, *Parallel Logic Programming in Parlog*, Addison-Wesley, 1987.

[3] M. Hanus and R. Sadre, A Concurrent Implementation of Curry in Java, *Workshop on Parallelism and Implementation Technology for CLP Languages*, New York, Oct. 1997

[4] S. Haridi and S. Janson, Kernel Andorra Prolog and its computational model, in *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[5] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[6] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, Springer-Verlag, 1994.

[7] J.W. Lloyd, *Foundation of Logic Programming*, Springer-Verlag, New-York, 1987.

[8] T..Muresan ,R. Potolea, N. Hodor, S. Moldovan and M. Nagy: Execution Mechanisms in Concurrent Logic Programming, At *International Conference on Technical Informatics*, Timisoara, Nov. 1994.

[9] A. Podelski and G. Smolka, Operational Semantics of Constraint Logic Programs with Coroutining, In *Proceedings of International Conference of Logic Programming*, MIT Press, June 1995.

[10] V.A. Saraswat, Compiling cp(↓,|,&) on top of Prolog, *Technical Report*, CMU-CS-87-174, Carnegie Mellon University, Oct. 1987.

[11] V.A. Saraswat, *Concurrent Constraint Programming*, MIT Press, ACM Doctoral Dissertation Award and Logic Programming Series, 1993.

[12] E. Shapiro, editor. *Concurrent Prolog*, vol 1 and 2. MIT Press, Dec. 1987.

[13] G. Smolka, *The Oz Programming Model*, In Jan van Leeuwen, editor, Computer Science Today, LNCS 1000, Springer-Verlag, 1995.