Residual SLDNF in CLP languages

Imperial College Research Report No. DoC 95/18

K J Dryllerakis

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London, SW7 2BZ, UK
Email: kd@doc.ic.ac.uk

October 1995
(Originally published May 1992)

Abstract

This report gives an outline of the theoretical and practical framework for the implementation of a restricted form of residual SLDNF in languages falling under the general CLP scheme ([JLM86]). The work is related to partial evaluation of goals in logic programs and to constructive negation ([Cha88]). In particular we implement constructive negation for arithmetic relations in the domain of real numbers under CLP($\mathcal{R}$).

Keywords

constructive negation, residual SLDNF, constraint logic programming, negation as failure, CLP(R)

1 The CLP Language Scheme

CLP is a general programming scheme. Different programming languages arise when a specific structure is specified. The best known of these languages is CLP($\mathcal{R}$) using as external structure the set of Real Numbers. The idea behind the scheme is to reason directly in the intended interpretation and not on all possible interpratations as normally happens on logic programming languages. The scheme claims to encompass all current logic programing languages as subcases. Prolog can be thought of as CLP($\mathcal{H}$) where $\mathcal{H}$ is the Herbrand Universe i.e. the set of syntactic objects of the language. A thinker can claim that adopting this point of view little relation remains to first order logic.

It is standard practise for the existing CLP languages to work on two -instead of one- domains: the domain of computation (e.g. the real numbers) and the herbrand universe of syntactic objects. That way, CLP(X) languages behave like prolog when no constraints are used. This gives a strongly typed language where terms belong either to the external domain or are simply herbrand terms.

The external structure contains a domain of discourse (we can always think of the real numbers as a good example) and a set of relations and functions that are to have a specific meaning (the interpreted functors). The relations of the external domain constitute the *constraints* and are handled directly by a constraint solver.

A constraint logic program is a set of *clauses* of the form

$$A \leftarrow \{C_1 \& \ldots \& C_n\}, B_1, \ldots, B_m$$

where A is an atom, $C_1, \ldots, C_n, n \geq 0$ are constraints and $B_1, \ldots, B_m, m \geq 0$ are atoms. A *goal* is a clause without a head

$$\leftarrow \{C_1 \& \ldots \& C_n\}, B_1, \ldots, B_m$$

A *derivation step* is the transition from a goal

$$\leftarrow \{C_1 \& \ldots \& C_n\}, B_1, \ldots, B_k, \ldots, B_m$$

where $\{C_1 \& \ldots \& C_n\}$ is a satisfiable conjuction of constraints (the current collected constraint set) and B_k is a selected atom or constraint to

either if B_k is a constraint and $\{C_1 \& \ldots \& C_n \& B_k\}$ is satisfiable

$$\leftarrow \{C_1 \& \ldots \& C_n \& B_k\}, B_1, \ldots, B_{k-1}, B_{k+1}, \ldots, B_m$$

or if B_k is an atom for which a clause

$$R \leftarrow E_1, \ldots, E_l$$

exist and the set $\{C_1 \& \ldots \& C_n \& \{R = D_k\}\}$ where $\{R = D_k\}$ is the set of constraints equating the arguments of R and B_k is satisfiable

$$\leftarrow \{C_1 \& \ldots \& C_n \& \{R = D_k\}\}, B_1, \ldots, B_{k-1}, B_{k+1}, \ldots, B_m$$

A *computation rule* determines the selection process of an atom at each derivation step. A *search tree* for a goal G to be the tree which has as root G and children of nodes are obtained by a derivation step applied to the parent. A *derivation* of a goal G is a branch in its search tree.

2 Viewing Constraints as residues

The result of a succesfull computation in CLP() languages is a set of constraints. Equality between terms is also viewed as a constraint (making unification just a subcase of constraints). These constraints can be viewed as residues of the final answer. What is more, since the semantics of the residue constraints are clearly defined (on the external structure) we can also compute their complement. The inderect handling of constraints in CLP languages, permits the easy implementation

of constructive negation with repsect to the domain-specific terms (e.g. to arithmetic terms for the domain of real numbers).

First of all let as suppose that for a goal G the search tree is finite and fully expanded. Each leaf node containts a set of collected constraints (which is eventually the answer to a sucessfull computation) which we will call H_i where

$$H_i = \{C_{i1} \& \ldots \& C_{il}\}.$$

According to first order logic the original goal G is logically equivalent to the disjunction:

$$G \equiv \bigvee_i \{C_{i1} \& \ldots \& C_{il}\} \equiv \bigvee_j H_j$$

3 Defining Negation

3.1 Negation as Failure

The semantics of negation as failure have been thorougly analysed in many works. If the goal is ground negation as failure acts simply as a test to the provability of the goal. If the goal contains free variables then we practice what is generally called unsafe negation which will return no binding for the free variables and may result to loss of corectness.

3.2 Constrcutive Negation

Suppose that after execution of the goal G we reduce it to the disjunction:

$$G \equiv \bigvee_j H_j$$

By *not* G we expect :

$$not\ G \equiv \bigwedge_i \{not\ C_{i1} \vee \ \ldots \ \vee not\ C_{il}\}$$

Remember now that any n-ary constraint (relation on D^n) has a *definite* meaning: the set of n-tuples from D^n for which the relation is true. Therefore the complement of this relation is just the set of tuples for which the relation is false. When working on a particular domain it is not difficult to implement the complement of a relation. Take for example the relation $\geq/2$ in real numbers. It is obvious that the complementary relation is $</2$. If this is the case

$$not\ C \quad \equiv \quad \overline{C}.$$

So, we can rewrite the negation for goal G as

$$not\ G \quad \equiv \quad \bigwedge_i \{\overline{C}_{i1} \vee \ \ldots \ \vee \overline{C}_{il}\}$$

As it can be easily seen this technique only applies to goals that are reduced to constraints on the domain of discource and not as equalities to the herbrand universe. In that case the variables appearing on the goal must be typed (at the time of execution of the goal) as domain variables (e.g. for the real numbers the variables must be arithmetic terms).

3.3 Conclusion

The above results are not in *any* way astonishing. Since we are working directly in the intended interpretation, all relations containing solely domain terms are semanticly reduced to relations on the (fixed) domain; they are nothing more but sets of n-tuples. The negation of any relation is just the complement of its truth set and can be easily calculated in the intended interpretation.

In what it is to follow we implement constructive negation for the CLP($\mathcal{R}$) language where the external domain is that of real numbers. The results are easily extensible to any external structure and can be readily implemented on any CLP(X) language.

4 Implementation of Generalised Negation

4.1 General Algorithm

The implementation of an extended version of negation incorporating constructive negation for domain specific relations is based on the following algorithm[1]:

```
if Goal is ground then
        execute negation as failure and
        END
if Goal contains non-domain variables then
        notify user,
        execute (unsafe) negation as failure and
        END
if Goal contains only free domain-variables then
        execute constructive negation and
        END
```

4.2 Negation as failure

First we need to implement negation as failure in the standard way:

```
do_nbf(Goal):-
        call(Goal),
        !,
        fail.
do_nbf(Goal).
```

4.3 Constructive Negation

Now, we need to implement constructive negation. The main idea is the following:

```
do_negation(Goal):-
        findallsolutions(Goal,SolutionSpace),
        complementof(SolutionSpace,ComplementSpace),
        constrain(Goal,ComplementSpace).
```

As it can be easily seen special predicates will be needed to find the free variables of the *Goal* and the find all possible solutions. Since the answer of such predicates will be in the form of constraints which are internal to the language we will also resort to metalevel facilities to dump constraints to a readable form and hopefully invert them. The implementation of the predicates `find_all/3`, `varsin/2`, `\==/2`, `<>`, `copy/2` and `copy_replace/4` can be found in the Technical Report: *Implementing System Predicates in CLP($\mathcal{R}$).*

When reading CLP($\mathcal{R}$) code keep in mind that constraints always accompany the variables in the calculation of a Goal and the only way we can access the constraints is either using the `dump` predicate or `assert/1`.

4.3.1 Top level predicate

We start by defining the top level unary predicate do_negation/1 which takes as input a term with free variables being domain variables (the checking can be insured by a CLP test checking for

[1] We adopt the convention that CLP($\mathcal{R}$) programs are typeset in `typewriter` font whereas general CLP algorithms in sans-serif

constraints on the variable e.g. `arithmetic/1` in CLP($\mathcal{R}$)). It works like that: construct a list of all variables in the Goal. Note that these variables can already be constrained i.e. the current constraint list for the variables of Goal might not be empty. Then, find all the possible solutions of the Goal and place them in a list of variables and add all the constraints to these new variables to the current constraint list. Take these variables and create a set of new variables which are constrained in the complementary way. Finally, constraint the variabes of Goal by equating them with the constrained variables.

```
do_negation(Goal):-
        varsin(quote(Goal),Vars),
        find_all(Vars,Goal,SolutionList),
        inverse_constraint_list(SolutionList,InvertedList),
        constraint_vars(Vars,InvertedList).
```

Note how much we rely on the constraint solver to keep track of inconsistencies. Also note the use of `quote/1` which is the only way we can use to get hold of the domain functions as normal (uniterpreted) functions (and thus $f(X)$ will not be returned as domain term i.e. a domain variable.

4.3.2 Inverse a list (of lists) of constrained variables

To inverse the list of (lists of) constrained variables we use

```
inverse_constraint_list([],[]).
inverse_constraint_list([X|Xs],[XX|XXs]):-
        complement_of(X,XX),
inverse_constraint_list(Xs,XXs).
```

4.4 Implementing the complement_of/2

The relation complement_of/2 should take a list of variables and return a copy of this list with fresh variables having the complementary constraints. We require a list of variables rather than the complement of a single variable because variables may be interconnected (e.g. `X<Y`). Note that in effect we are creating a set of fresh variables and augmenting the current constraint set with the proper cnstraints on the new variables. This implementation of this relation relies heavily on the language used and can also be an internal (rather that an external metaprogramming) tool. Also the mode in which used is important: the second argument must be a variable uncostrained term. So, here is the implementation of complement_of/2 in CLP($\mathcal{R}$).

First check for correct usage:

```
complement_of(X,Y):-
        nonvars(Y),     % Y is not a list of vars!
        write("Wrong Usage of complement_of/2:"),
        writeln("second arg is not a list of vars"),
        !, fail.
complement_of(X,Y):-
        arithmetics(Y),% Y is a list that contains arithmetic terms
        write("Wrong Usage of complement_of/2:"),
        writenl("second arg is arithmetic"),
        !,fail.
```

Then check if domain constants are part of the incoming list. In that case inequality must be returned. Since there is no such constraint, we can simulate it logically as $X \neq Y \leftarrow (X < Y \vee X > Y)$ or as a constraint $abs(X - Y) > 0$. We prefer the first logical definition:

```
complement_of(X,Y):-
        find_nonvar_in(X,Num,Y,Rnum),
        (Rnum<Num ; Rnum>Num).
```

```
find_nonvar_in([X|Xs],X,[Y|Ys],Y):-
        nonvar(X), arithmetic(X).
find_nonvar_in([X|Xs],X1,[Y|Ys],Y1):-
        find_nonvar_in(Xs,X1,Ys,Y1).
```

Having ensured correct usage and the existance of constraints we continue with the meaty part of the implementation:

```
complement_of(X,Y):-
        asserta('   ##INV'(X)),% Dump Constraints on X
        retract('   ##INV'(M)),% If this succeeds no constrains on X!
        !,
        fail.
complement_of(X,Y):-
        arithmetics(X), % test already performed
        list_of_reals(X,Y),
        retract('   ##INV'(X):- CX),% CX are the quoted constraints on X
        r_c_on(X,CX,Y,CY),
        call(eval(CY)).
r_c_on(X,(A,B),Y,(C;D)):-
        arithmetics(Y), % test already performed
        !,
        r_c_on(X,A,Y,C),
        r_c_on(X,B,Y,D).

r_c_on(X,A,Y,C):-
        copy_replace(X,quote(A),Y,C1),
        inv_co(eval(C1),quote(C)).

inv_co(abs(X-Y)>0,X=Y).
inv_co(X=Y,X<>Y).
inv_co(X>Y,X<=Y).
inv_co(X<Y,X>=Y).
inv_co(X<=Y,X>Y).
inv_co(X>=Y,X<Y).
inv_co(real(X),_):-fail.
```

Some utility predicates were used in the above piece of code so here is their implementation:

```
arithmetics([X]):-arithmetic(X).
arithmetics([X|Xs]):-
        arithmetic(X),
        arithmetics(Xs).
nonvars([X]):- nonvar(X).
nonvars([X|Z]):- nonvar(X),nonvars(Z).
list_of_reals([],[]).
list_of_reals([X|Xs],[Y|Ys]):-
        real(Y),
        list_of_reals(Xs,Ys).
```

4.4.1 Implementating the variable constrainer

Finally we need to implement the variable constrainer, a predicate that will constraint the value of a variable according to the constraints of a list of variables. This is just the logical conjuction of the constraints but all handling of inconsistancies is done by the constraint solver. Then input is an unbound variable and a list of domain-constrained variables. At the end the input variable has all constraints of the list of variables.

```
constraint_vars(Vars,[]).
constraint_vars(Vars,[LVars|MoreVars]):-
        constraint_all_vars(Vars,LVars),
        constraint_vars(Vars,MoreVars).
constraint_all_vars([],[]).
constraint_all_vars([X|Xs],[CX|CXs]):-
        X=CX,
        constraint_all_vars(Xs,CXs).
inverse_constraint_list([],[]).
inverse_constraint_list([X|Xs],[XX|XXs]):-
        complement_of(X,XX),
        inverse_constraint_list(Xs,XXs).
```

5 A listing of the CLP($\mathcal{R}$) source.

```
::-dynamic('   ##INV',1).
::-op(60,fx,do_not).
::-op(60,fx,\+).
\+ X :- do_not X.
::-prot(\+,1).
do_nbf(Goal):-
        call(Goal),
        !,
        fail.
do_nbf(Goal).
::-prot(do_nbf,1).
do_not(Goal):-
        ground(Goal),
        !,
        do_nbf(Goal).
do_not(Goal):-
        varsin(quote(Goal),Vars),
        !,
        (arithmetics(Vars) ->
                find_all(Vars,Goal,SolutionList),
                inverse_constraint_list(SolutionList,InvertedList),
                constraint_vars(Vars,InvertedList)
        ;
                write("Your free variables are not of arithmetic type"),nl,
                write("If you think it should be include a real(Var) predicate")
,nl,
                write("I will try unsafe negation"),nl,
                do_nbf(Goal)
        ).
::-prot(do_not,1).
constraint_vars(Vars,[]).
```

```
constraint_vars(Vars,[LVars|MoreVars]):-
        constraint_all_vars(Vars,LVars),
        constraint_vars(Vars,MoreVars).
constraint_all_vars([],[]).
constraint_all_vars([X|Xs],[CX|CXs]):-
        X=CX,
        constraint_all_vars(Xs,CXs).
inverse_constraint_list([],[]).
inverse_constraint_list([X|Xs],[XX|XXs]):-
        complement_of(X,XX),
        inverse_constraint_list(Xs,XXs).

complement_of(X,Y):-
        nonvars(Y),     % Y is not a list of vars!
        write("Wrong Usage of complement_of/2
second arg is not a list of vars"),nl,
        !,
        fail.

complement_of(X,Y):-
        arithmetics(Y),% Y is a list that contains arithmetic terms
        write("Wrong Usage of complement_of/2
second arg is arithmetic"),nl,
        !,
        fail.
complement_of(X,Y):-
        find_nonvar_in(X,Num,Y,Rnum),
       (Rnum<Num ; Rnum>Num).

find_nonvar_in([X|Xs],X,[Y|Ys],Y):-
        nonvar(X), arithmetic(X).
find_nonvar_in([X|Xs],X1,[Y|Ys],Y1):-
        find_nonvar_in(Xs,X1,Ys,Y1).
complement_of(X,Y):-
        asserta('   ##INV'(X)),   % Dump Constraints on X
        retract('   ##INV'(M)),   % If this succeeds no constraints on X!
        !,
        fail.
complement_of(X,Y):-
        arithmetics(X), % test already performed
        list_of_reals(X,Y),
        retract('   ##INV'(X):- CX),      % CX are the quoted constraints on X
        r_c_on(X,CX,Y,CY),
        call(eval(CY)).
r_c_on(X,(A,B),Y,(C;D)):-
        arithmetics(Y), % test already performed
        !,
        r_c_on(X,A,Y,C),
        r_c_on(X,B,Y,D).

r_c_on(X,A,Y,C):-
        copy_replace(X,quote(A),Y,C1),
        inv_co(eval(C1),quote(C)).
```

```
inv_co(abs(X-Y)>0,X=Y).
inv_co(X=Y,X<>Y).
inv_co(X>Y,X<=Y).
inv_co(X<Y,X>=Y).
inv_co(X<=Y,X>Y).
inv_co(X>=Y,X<Y).
inv_co(real(X),_):-fail.

arithmetics([X]):-arithmetic(X).
arithmetics([X|Xs]):-
        arithmetic(X),
        arithmetics(Xs).
nonvars([X]):- nonvar(X).
nonvars([X|Z]):- nonvar(X),nonvars(Z).
list_of_reals([],[]).
list_of_reals([X|Xs],[Y|Ys]):-
        real(Y),
        list_of_reals(Xs,Ys).
::-prot(constraint_vars,2),prot(constraint_all_vars,2),
        prot(inverse_constraint_list,2),prot(complement_of,2),
        prot(find_nonvar_in,4),prot(r_c_on,4),prot(inv_co,2),
        prot(arithmetics,1),prot(nonvars,1),prot(list_of_reals,2).
```

6 A listing of system predicates required (CLP($\mathcal{R}$) source).

```
%
?-printf("\n CLP(R) System Extension (MODULE system):\n ",[]).

?-dynamic('  find all',1), dynamic('  all found',3).

?-op(37,xfx,\==).       % Not unifiable
?-op(900,xfx,--).       % for difference lists

%
?-printf("\r writeseq/2",[]).
writeseq(X,Y):-
        printf("\r                                                \r",[]),
        printf(X,Y).
::-prot(writeseq,2).

?-writeseq("findall/3",[]).
find_all(Te,En,Li):-
        asserta('  find all'([])),
        call(En),
        asserta('  find all'('  wrap'(Te))),
        fail;
        '  all found'([],Li).

'  all found'(SoFar,List):-
        '  ##get retract'(Item),
        !,
        '  all found'(Item,SoFar,List).

'  all found'([],List,List).
```

```
'  all found'('  wrap'(Te),SoFar,List):-
        '  all found'([Te|SoFar],List).

'  ##get retract'('  wrap'(X)):-
        retract('  find all'('  wrap'(X)):-W),
        realterm(X), % Do we really want to restrict X?
                %If this retract works
                % it means that X has constraints W i.e. it should be real
                % in that case the returned variable in find_all should be
                % real as well.
        call(eval(W)).
'  ##get retract'(X):-
        retract('  find all'(X)).
::-writeseq("realterm/1",[]).
realterm(X):-
        varsin(X,W),
        '  ##all real'(W),!.

realterm(X):-
        real(X).
'  ##all real'([X]):-
        real(X).
'  ##all real'([X|Xs]):-
        real(X),
        '  ##all real'(Xs).

::-prot(find_all,3).
::-prot('  find all',1).
::-prot('  all found',3), prot('  all found',2).
::-prot('  ##get retract',1).
::-prot('  ##all real',1),prot(realterm,1).

::-writeseq("\\==/2",[]).
A \== A:-
        !,
        fail.

A \== B.
::-prot('\==',2).
::-writeseq("varsin/2",[]).
varsin(Term,Vars):-
        '  ##varsin'(Term,Vs,[]),
        '  ##removedoubles'(Vs,Vars).

'  ##varsin'(Term,[Term|Rest],Rest):-
        var(Term),!.

'  ##varsin'(Term,List,Rest):-
        (functor(Term,_,N);
         atomic(Term),
         N=0),
        '  ##varsin'(N,Term,List,Rest).
```

```
'  ##varsin'(N,Term,S,S):-
        N=0,!.

'  ##varsin'(N,Term,S0,S):-
        arg(N,Term,Arg),
        '  ##varsin'(Arg,S0,S1),
        M=N-1,
        '  ##varsin'(M,Term,S1,S).
'  ##removedoubles'(X,Y):-'  ##removedoubles'(X,Y,[]).

'  ##removedoubles'([],Y,Y).
'  ##removedoubles'([X|Rest],List,Built):-
        ('  ##inlist'(X,Built) ->
        '  ##removedoubles'(Rest,List,Built);
        '  ##removedoubles'(Rest,List,[X|Built])
        ).

'  ##inlist'(X,[Y|Z]):- X == Y.
'  ##inlist'(X,[Y|Z]):-
        '  ##inlist'(X,Z).
::-prot(varsin,2).
::-prot('  ##varsin',2).
::-prot('  ##varsin',3).
::-prot('  ##varsin',4).
::-prot('  ##removedoubles',2).
::-prot('  ##removedoubles',3).
::-prot('  ##inlist',2).
?-writeseq("unify/2",[]).
unify(X,X).

::-prot(unify,2).
?-writeseq("<>/2",[]).

?-op(40,xfx,<>).

X <> Y :-
        (X<Y;
        Y<X).

::-prot('<>',2).
::-dynamic('  ##????',1),prot('  ##????',1).

?-writeseq("copy/2",[]).
copy(Term1,Term2):-
        assert('  ##????'(Term1)),
        (retract('  ##????'(Term2):-W);
        retract('  ##????'(Term2))).
::-prot(copy,2).
?-writeseq("copy_replace",[]).

copy_replace(X,Term1,Y,Term2):-
        varsin(Term1,Vars1),
        copy(Term1,Term2),
        varsin(Term2,Vars2),
```

```
        '  ##bind special list'(X,Vars1,Y,Vars2).

'  ##bind special list'(X,[],Y,[]).
'  ##bind special list'(X,[V1|V1s],Y,[V2|V2s]):-
        ('  ##in list ret'(X,V1,Y,RetY) ->
                unify(RetY,V2);
                unify(V1,V2)),
        '  ##bind special list'(X,V1s,Y,V2s).

'  ##in list ret'([X|Xs],V1,[Y|Ys],Y):- X == V1.
'  ##in list ret'([X|Xs],V1,[Y|Ys],RetY):-
        '  ##in list ret'(Xs,V1,Ys,RetY).

::-prot(copy_replace,4).
::-prot('  ##in list ret',4),prot('  ##bind special list',4).

'  ##system loaded'.
::-prot('  ##system loaded',0).
::-writeseq("done.\n",[]).
```

References

[Cha88] David Chan. Constructive negation based on the completed database. Technical report, European Computer–Industry Research Centre, 1988.

[JLM86] J Jaffar, J-L Lassez, and M. J. Maher. A logic programming language scheme. In D DeGroot and G Lindstrom, editors, *Logic Programming: Relations, Functions and Equations*. Prentice-Hall, 1986.