Entailment and projection for CLP(B) and CLP(Q) in SICStus Prolog

Fred Mesnard

Iremia, Université de la Réunion
15, avenue René Cassin - BP 7151 -
97 715 Saint Denis Messag. Cedex 9 France
E-mail: fred@univ-reunion.fr
Phone: (+262) 93 82 82 Fax : (+262) 93 82 60

Abstract

The idea of using the CLP framework to reason about CLP programming leads to not only an elegant theory but also to practical and useful applications. Most works rely on abstract top-down or bottom-up interpreters to finitely compute the semantics of abstract programs. A few basic operations, e.g. entailment and projection, are quite common in the structure of such interpreters. This paper presents the definition, specification and implementation of five relations, namely *satisfiable*, *entail*, *equivalent*, *project* and *copy* for the boolean and rational constraint solvers of SICStus Prolog v3-5.

1 Introduction

The idea of using the CLP framework to reason about CLP programming leads to not only an elegant theory [3, 5] but also to practical and useful applications. Some of them are described in [10] for automatic proof of program properties, in [1] for inference of inter-argument relations and in [8, 9] for termination analysis.

Most works rely on an abstract top-down or bottom-up interpreter to finitely compute the semantics of an abstract program P_a, characterizing some properties of the original concrete program P.

A few basic relations, e.g. entailment and projection, are quite common in the structure of such interpreters. This paper presents the definition, specification and implementation[1] for the SICStus Prolog boolean and rational constraint solvers of five relations, namely *satisfiable*, *entail*, *equivalent*, *project* and *copy*.

We use the non-ground approach for meta-constraint programming. The semantics is surely less clean, but we think that this choice leads to more efficient code. We choose SISCtus Prolog v3-5. This Prolog system is available on many platforms and comes with a

[1] which may still contain some errors...

large number of packages in its library including a boolean solver [2] and a rational solver [6].

We organize the report as follows. Section 2 defines our notations and the five basic relations we are interested in. The purpose of section 3 is to identify the conditions which allow a clean implementation of the entailment check. Section 4 gives the specifications of our logic procedures. The next two sections list the Prolog code for our basic relations. After the conclusion, we present an implementation of the computation of the convex hull of two rational subspaces and include the code which completes the projection for CLP(Q).

2 Preliminaries

We try to stick to the notations and the conventions introduced in [7]. The set of variables of a term t is denoted $var(t)$. We say that θ is a *solution* of the constraint c if θ is a valuation s.t. $\mathcal{D} \models c\theta$. Let c_1 and c_2 be two constraints. We write $c_1 \models c_2$ as a shorthand for $\mathcal{D} \models \forall[c_1 \rightarrow c_2]$.

We consider two domains in the paper : the booleans and the rational numbers.

A boolean constraint c is coded as usual as a CLP(B) SICStus Prolog term using only the constants `0` and `1`, the variables, the negation, and the binary logical operators `*`, `+`, etc. available for the boolean solver. We disallow explicitly quantified variables (universally or existentially) but we internally uses existential quantification for variable elimination in projection.

A linear rational constraint $c_1 \wedge \cdots \wedge c_n$ is coded as a Prolog list `[C1,...,Cn]` where `Ci` is the CLP(Q) SICStus Prolog representation of c_i.

Example 2.1 The boolean constraint $(x \leftrightarrow y) \wedge \exists z((u + z) \rightarrow v)$ is coded `(X=:=Y)*(Z^ (U+Z =< V))`. The rational constraint $x \geq 2y + 1 \wedge z \leq 0$ is coded `[X >= 2*Y+1, Z =< 0]`.

An *environment* is a term $env(\tilde{x}, t)$ where t is a non-ground term coding a constraint which we now confuse with t. Let $\mathcal{D}$ be the domain of computation. The meaning of an environment is defined by:

$$||env(\tilde{x}, t)|| = \{\tilde{x}\theta \mid \mathcal{D} \models t\theta\}$$

Example 2.2 There is a priori no relationship between $var(t)$ and $\tilde{x}$. We have for instance: $||env(x, x = x)|| = ||env(x, y = z)|| = ||env(x, x = y)|| = \mathcal{D}$

We present five basic relations which should be useful for building constraint meta-programs. The relations are defined with their "minimal intuitive" semantics.

- $satisfiable(t)$ iff $\mathcal{D} \models \tilde{\exists} t$
- $entail(\tilde{x}, s, \tilde{y}, t)$ iff $||env(\tilde{x}, s)|| \subseteq ||env(\tilde{y}, t)||$
- $equivalent(\tilde{x}, s, \tilde{y}, t)$ iff $||env(\tilde{x}, s)|| = ||env(\tilde{y}, t)||$
- $project(\tilde{x}, s, \tilde{y}, t)$ iff $||env(\tilde{x}, s)|| = ||env(\tilde{y}, t)|| \wedge var(t) \subseteq \tilde{y}$
- $copy(\tilde{x}, s, \tilde{y}, t)$ iff $||env(\tilde{x}, s)|| = ||env(\tilde{y}, t)|| \wedge var(env(\tilde{x}, s)) \cap var(env(\tilde{y}, t)) = \phi$

Before we switch to the specifications and the programs for these relations, let us precise the *entail* relation.

3 Refining the *entail* relation

We'd like to restate the *entail* relation in a pure constraint logic language. More precisely, we aim at finding the *weaker* precondition on $\tilde{x}$, $var(s)$, $\tilde{y}$ and $var(t)$ such that we have $entail(\tilde{x}, s, \tilde{y}, t)$ iff $\tilde{x} = \tilde{y} \models s \rightarrow t$. We have two reasons in mind. First, we hope to ensure correctness of our implementation. Second, we want to avoid too many calls to the costly logic procedure `project`/4.

One may first prove the following equality:

$$\{\tilde{x}\sigma | \mathcal{D} \models t\sigma\} = \{\tilde{x}\theta | \mathcal{D} \models \exists_{-\tilde{x}} t\theta\}$$

Next,

Proposition 3.1 we have the following properties:

P1 If *true* then
$||env(\tilde{x}, s)|| \subseteq ||env(\tilde{y}, t)||$ implies $\tilde{x} = \tilde{y} \wedge \exists_{-\tilde{x}} s \models \exists_{-\tilde{y}} t$.

P2 If $\tilde{x} \cap \tilde{y} = \phi$ then
$\tilde{x} = \tilde{y} \wedge \exists_{-\tilde{x}} s \models \exists_{-\tilde{y}} t$ implies $||env(\tilde{x}, s)|| \subseteq ||env(\tilde{y}, t)||$.

P3 If *true* then
$\tilde{x} = \tilde{y} \wedge \exists_{-\tilde{x}} s \models \exists_{-\tilde{y}} t$ implies $\tilde{x} = \tilde{y} \wedge s \models \exists_{-\tilde{y}} t$.

P4 If $var(env(\tilde{x}, s)) \cap \tilde{y} = \phi$ then
$\tilde{x} = \tilde{y} \wedge s \models \exists_{-\tilde{y}} t$ implies $\tilde{x} = \tilde{y} \wedge \exists_{-\tilde{x}} s \models \exists_{-\tilde{y}} t$.

P5 If $var(s) \subseteq \tilde{y}$ then
$\tilde{x} = \tilde{y} \wedge s \models \exists_{-\tilde{y}} t$ implies $\tilde{x} = \tilde{y} \wedge s \models t$.

P6 if *true* then
$\tilde{x} = \tilde{y} \wedge s \models t$ implies $\tilde{x} = \tilde{y} \wedge s \models \exists_{-\tilde{y}} t$.

Hence, if we gather all the previous implications,

Corollary 3.1 we have:

P7 If $var(env(\tilde{x}, s)) \cap \tilde{y} = \phi \wedge var(t) \subseteq \tilde{y}$ then
$entail(env(\tilde{x}, s), env(\tilde{y}, t))$ iff $\tilde{x} = \tilde{y} \models s \rightarrow t$.

P8 If $\tilde{x} \cap \tilde{y} = \phi \wedge var(s) \subseteq \tilde{x} \wedge var(t) \subseteq \tilde{y}$ then
$equivalent(env(\tilde{x}, s), env(\tilde{y}, t))$ iff $\tilde{x} = \tilde{y} \models s \leftrightarrow t$.

Property **P7** tells us that we don't have to project both environments to check entailment. Note that it is a quite usual situation. An abstract interpreter often maintains a data structure recording the partial semantics of each procedure of the abstract program in a projected form. Then, when a new solution is computed, before updating the data structure, one checks whether the solution is already present. So it is not necessary to project the new solution to perform the entailment check. On the other hand, the equivalent test only applies to two projected environments.

4 Specifications of our implementations

We try to give the "logical" specifications of our implementation with respect to the five relations we propose in section 2. The specifications are common to both implementations. However, as we use a non-ground representation and meta-logic built-in procedures, we have to deal with mode information. So the specifications can't be "completely" logical.

Let $\tilde{x}$ and $\tilde{y}$ be two sequences of distinct variables coded as the Prolog lists `Xs` and `Ys`. Let s and t be two terms representing two constraints coded as the Prolog terms `S` and `T`.

- `satisfiable(+T)` iff $satisfiable(t)$
- `copy(+Xs,+S,-Ys,-T)` iff $copy(\tilde{x}, s, \tilde{y}, t)$

Note that the meaning of *copy* is only an approximation of its implementation using `copy_term/2`: `copy(Xs,S,Ys,T):-copy_term(env(Xs,S),env(Ys,T)).` This logic procedure generates the *fresh* copy we *do* need.

- `project(+Xs,+S,-Ys,-T)` iff
  $project(\tilde{x}, s, \tilde{y}, t) \wedge satisfiable(s) \wedge var(env(\tilde{x}, s)) \cap var(env(\tilde{y}, t)) = \phi$

Our `project(+Xs,+S,-Ys,-T)/4` procedures check the consistency of s and deliver a fresh copy of the projection of s on $\tilde{x}$. For the last two relations, the results of section 3 give some preconditions:

- If $var(env(\tilde{x}, s)) \cap \tilde{y} = \phi \wedge var(t) \subseteq \tilde{y}$ then
  `entail(+Xs,+S,+Ys,+T)` iff $entail(\tilde{x}, s, \tilde{y}, t)$
- If $\tilde{x} \cap \tilde{y} = \phi \wedge var(s) \subseteq \tilde{x} \wedge var(t) \subseteq \tilde{y}$ then
  `equivalence(+Xs,+S,+Ys,+T)` iff $equivalence(\tilde{x}, s, \tilde{y}, t)$

5 An implementation in CLP(B)

We give our Prolog code for CLP(B) and some hints which may help to understand it. The logic procedure `copy/4` is defined in section 4.

```
check(G):- \+ \+ call(G).

:-use_module(library(clpb)).

satisfiable(T):-check(sat(T)).
```

The double negation prevents any binding on the variables of `T`. For the `entail/4` and `equivalent/4` logic procedures, we do rely on the `taut/2`, *once fixed with the two patches for* `taut/2` *available from SICStus.*

```
entail(Xs,S,Ys,T):-check( (Xs=Ys,taut(S =< T,1)) ).

equivalent(Xs,S,Ys,T):-check( (Xs=Ys,taut(S =:= T, 1)) ).
```

The basic idea for projection is first to simplify the constraint `S` using the underlying boolean constraint solver. Second we get the resulting constraints using `call_residue/2`, normalize the sequence of terms `X1s` in a sequence of distinct variables `X2s`, copy the projected environment in the blackboard and fail to forget the bindings on `Xs`. Third, we get the copy, compute the set of free variables in `T1` and eliminate them using the existential quantification available in boolean solver of SICStus Prolog. We assume that the boolean solver never uses explicit existential quantification in the answers it gives.

```
project(Xs,S,_,_):-
          sat(S),bb_put(copy,Xs),
          call_residue(bb_delete(copy,X1s),C),gather(C,Cs),
          normalize(X1s,X2s,[],Cs,C2s),
          bb_put(copy,env(X2s,C2s)),
          fail.
project(_,_,Ys,T):-
          bb_delete(copy,env(Ys,T1)),
          free_vars(T1,Ys,FreeVars),
          eliminate(FreeVars,T1,T).

gather([],1).
gather([_-sat(C)|Cs],T):-gather(Cs,C,T).

gather([],C,C).
gather([_-sat(C1)|Cs],C2,T):-gather(Cs,C2*C1,T).

normalize([],[],_,Cs,Cs).
normalize([X|Xs],[Y|Ys],Vars,C1s,C2s):-
          var(X),member_var(Vars,X),!,normalize(Xs,Ys,Vars,(Y=:=X)*C1s,C2s).
normalize([X|Xs],[X|Ys],Vars,C1s,C2s):-
          var(X),!,normalize(Xs,Ys,[X|Vars],C1s,C2s).
normalize([X|Xs],[Y|Ys],Vars,C1s,C2s):-
          % ground(X),
          (X=0 -> Z= ~Y ;  Z=Y),
          normalize(Xs,Ys,Vars,Z*C1s,C2s).

member_var([Y|_],X):-X==Y.
member_var([_|Ys],X):-member_var(Ys,X).

free_vars(Term,Vars,FreeVars):-fv(Term,Vars,[],FreeVars).

fv(X,Vars,FV,FV):-var(X), (member_var(Vars,X) ; member_var(FV,X)),!.
fv(X,_,FV,[X|FV]):-var(X),!.
fv(X,_,FV,FV):-ground(X),!.
fv(~T,Vars,FV1,FV2):-!,fv(T,Vars,FV1,FV2).
fv(T,Vars,FV1,FV2):-T=..[_,T1,T2],fv(T1,Vars,FV1,FV3),fv(T2,Vars,FV3,FV2).

eliminate([],T,T).
eliminate([X|Xs],T,U):-eliminate(Xs,(X^T),U).
```

6 An implementation in CLP(Q)

We give our Prolog code for CLP(Q).

```
:-use_module(library(clpq)).

satisfiable(T):-check(callc(T)).

callc([]).
callc([C|Cs]):-{C},callc(Cs).

entail(Xs,S,Ys,T):-check((Xs=Ys,callc(S),entail_conj(T))).

entail_conj([]).
entail_conj([C|Cs]):-entailed(C),entail_conj(Cs).
```

Note that if the intended domain of computation is the set of natural numbers and if the constraints follows the form `T1 =< T2`, `T1 = T2` or `T1 >= T2`, one may give a more precise definition of `entail_conj/1`:

```
entail_conj([]).
entail_conj([C|Cs]):-negate_nat(C,Cn),\+ {Cn},entail_conj(Cs).

negate_nat(A=<B, A>=B+1).
negate_nat(A=B,  (A>=B+1; A+1 =< B)).
negate_nat(A>=B, A+1 =< B) .
```

The procedure `equivalent/4` can be defined using `entail/4`:

```
equivalent(Xs,S,Ys,T):-entail(Xs,S,Ys,T),entail(Ys,T,Xs,S).
```

But if we know that the environments were previously projected and were not modified after the projection (e.g. permutation of two atomic constraints), then a syntactic check is all we need. We assume that two equivalent environments, after projection, have the same canonical form up to a renaming.

```
equivalent(Xs,S,Ys,T):-check((numbervars(S,0,_),numbervars(T,0,_),T==S)).
```

The code for projecting an environment follows the ideas of the boolean projection but relies on the procedure `linear:dump/3` (see the appendix). We first tried to transpose the schema defined in section 5. But for CLP(Q), it seems that the projection obtained with `call_residue/2` is not perfect.

```
project(Xs,S,_,_):-
          callc(S),
          normalize(Xs,X1s,X2s,[],Cons,Eqns),
          linear:dump(X1s,X2s,Cons),
          bb_put(copy,env(X2s,Eqns)),
          fail.
project(_,_,Ys,T):-
          bb_delete(copy,env(Ys,T)).

normalize([],[],[],_,Tail,Tail).
normalize([X|Xs],[Y|Ys],[Y|Zs],Vars,Tail,[Y=Z|Eqns]):-
          var(X),already_in(Vars,X-Z),!,
          normalize(Xs,Ys,Zs,Vars,Tail,Eqns).
normalize([X|Xs],[X|Ys],[Z|Zs],Vars,Tail,Eqns):-
          var(X),!,
```

```
        normalize(Xs,Ys,Zs,[X-Z|Vars],Tail,Eqns).
normalize([X|Xs],[Y|Ys],[Y|Zs],Vars,Tail,[Y=X|Eqns]):-
        % ground(X),
        normalize(Xs,Ys,Zs,Vars,Tail,Eqns).

already_in([Y-Z|_],X-Z):-Y==X.
already_in([_|Ys],X-Z):-already_in(Ys,X-Z).
```

7 Conclusion

In this paper, we present the definition, specification and implementation of five basic relations: *satisfiable*, *entail*, *equivalent*, *project* and *copy* for the boolean and rational constraint solvers of SICStus Prolog v3-5.

We hope it will be helpful to reseachers involved in the analysis of CLP with CLP. And it would be nice if our community could make useful meta-constraint programs publicly available to prove to our students and colleagues that semantic-based program analysis is not a virtual reality !

Acknowledgments

We use the macros defined by Steve Kelem in his `latex2pl` package to include Prolog programs. Andy King and Florence Benoy sent us their program described in [1] which became one of the starting points of this work. Christian HolzBaur kindly commented a first draft of the paper and allowed us to reproduce his code for the `linear:dump/3` procedure.

Appendix 1: `convex_hull/4`

Once we have projection, we can propose a logic procedure:

- If *satisfiable*$(s)\wedge$ *satisfiable*(t) then `convex_hull(+Xs,+S,+Ys,+T,-Zs,-U)` iff $env(\tilde{z},u)$ is the convex hull of $env(\tilde{x},s)$ and $env(\tilde{y},t)$

We point out that s and t denote any polyhedra (including points, lines, linesegments and polytopes). The trick (cf. [4, 1]) is to linearize the problem by introducing two parameters. Further, the code trivially generalizes to compute the hull over n objects.

```
convex_hull(Xs,Cxs,Ys,Cys,Zs,Czs):-
        scale(Xs,Cxs,S1,Vs,□,Cvs),
        scale(Ys,Cys,S2,Ws,Cvs,Cvws),
        add_vect(Vs,Ws,Us,Cvws,Cs),
        project(Us,[S1>=0,S2>=0,S1+S2=1|Cs],Zs,Czs).

add_vect([],□,[],Cs,Cs).
add_vect([X|Xs],[Y|Ys],[Z|Zs],C1s,C2s):-
        add_vect(Xs,Ys,Zs,[X+Y=Z|C1s],C2s).
```

```
scale(Xs,Cxs,S,Ys,Cws,Cyws):-
          copy_term(env(Xs,Cxs),env(Ys,Cys)),
          multis(Cys,S,Cws,Cyws).

multis([],_,Cs,Cs).
multis([C1|C1s],S,C2s,C3s):-
          multi(C1,S,C2),multis(C1s,S,[C2|C2s],C3s).

multi(C1,S,C2):-
          C1=..[OpRel,A1,B1],C2=..[OpRel,A2,B2],
          multExp(A1,S,A2),multExp(B1,S,B2).

multExp(X,_,X):-var(X),!.
multExp(-X,S,-Y):-!,multExp(X,S,Y).
multExp(N,S,N*S):-ground(N),!.
multExp(N*X,_,N*X):-ground(N),var(X),!.
multExp(A+B,S,C+D):-!,multExp(A,S,C),multExp(B,S,D).
multExp(A-B,S,C-D):-!,multExp(A,S,C),multExp(B,S,D).
multExp(X,_,_):-raise_exception(multExp(X,_,_)).  % oops !
```

Here is an example:

```
?- convex_hull([X,Y],[0=<X,X=<1,Y>=0,Y=<2],[X,Z],[2=<X,X=<3,-1=<Z,Z=<1],Vars,Cs).

Cs = [_A>=0,_A=<3,_B=<2,_A+2*_B=<5,_A+2*_B>=0,_B>=-1],
Vars = [_A,_B] ? ;

no
```

Appendix 2: `linear:dump/3`

We only reformat the code to save space, decreasing its readability (but not so much ..!).

```
/*
       linear:dump( +Target, ?NewVars, ?CodedAnswer)
where Target and NewVars are lists of variables of equal length and
CodedAnswer is the term representation of the projection of constraints
onto the target variables where the target variables are replaced by
the corresponding variables from NewVars.
*/
:- module(linear).

:- use_module(library(terms),[term_variables/2]).

:- use_module(library(assoc),
          [empty_assoc/1,get_assoc/3,put_assoc/4,assoc_to_list/2]).

dump(Target,NewVars,Constraints) :-
          (   ordering(Target),related_linear_vars(Target,All),
              nonlin_crux(All,Nonlin),project_attributes(Target,All),
              related_linear_vars(Target,Again),
              all_attribute_goals(Again,Gs,Nonlin),
              empty_assoc(D0),copy(Target/Gs,TmpCopy,D0,_),
              bb_put(copy,TmpCopy),fail
              ;bb_delete(copy,NewVars/Constraints)).
```

```
related_linear_vars(Term,Alls) :-
          term_variables(Term,Vs), empty_assoc(S0),
          related_linear_sys(Vs,S0,Sys),related_linear_vars(Sys,All,Vs),
          sort(All,Alls).

related_linear_sys([],S0,L0) :- assoc_to_list(S0,L0).
related_linear_sys([V|Vs],S0,S2) :-
          ( get_atts(V,class(C)) ->
              put_assoc(C,S0,C,S1)
              ;S1 = S0),
          related_linear_sys(Vs,S1,S2).

related_linear_vars([]) --> [].
related_linear_vars([S-_|Ss]) -->
          {class:allvars(S,Otl)},cpvars(Otl),related_linear_vars(Ss).

cpvars(Xs) --> {var(Xs)}, !.
cpvars([X|Xs]) --> ( {var(X)} -> [X] ; [] ),cpvars(Xs).

nonlin_crux(All,Gs) :- empty_assoc(D0),nonlin_crux(All,D0,_,Gs,[]).

nonlin_crux([],D0,D0) --> [].
nonlin_crux([V|Vs],D0,D2) -->
          ( {geler:get_atts(V,goals(G))} ->
               crux(G,D0,D1)
               ;[], {D1=D0}),
          nonlin_crux(Vs,D1,D2).

crux((A,B),D0,D2) --> crux(A,D0,D1), crux(B,D1,D2).
crux(run(Mutex,G),D0,D1) -->
          ( {nonvar( Mutex)} ->
               {D1=D0}
               ;{get_assoc(Mutex,D0,_)} ->
                    {D1=D0}
                    ;{put_assoc(Mutex,D0,Mutex,D1),
                     geler:transg(G,Gt,[])},crux(Gt)).

crux([]) --> [].
crux([G|Gs]) --> crux(G), crux(Gs).
crux(nonlin:{G}) --> [G].

all_attribute_goals([]) --> [].
all_attribute_goals([V|Vs]) -->
          dump_linear(V,toplevel),dump_nonzero(V,toplevel),
          all_attribute_goals(Vs).

copy(Term,Copy,D0,D1) :- var(Term),
          ( get_assoc(Term,D0,New) ->
              Copy = New,D1 = D0
              ;put_assoc(Term,D0,Copy,D1)).
copy(Term,Copy,D0,D1) :- nonvar(Term),
          functor(Term,N,A),functor(Copy,N,A),copy(A,Term,Copy,D0,D1).

copy(0,_,_,D0,D0) :- !.
copy(1,T,C,D0,D1) :- !,
```

```
          arg(1,T,At1),arg(1,C,Ac1),copy(At1,Ac1,D0,D1).
copy(2,T,C,D0,D2) :- !,
          arg(1,T,At1),arg(1,C,Ac1),
          copy(At1,Ac1,D0,D1),
          arg(2,T,At2),arg(2,C,Ac2),
          copy(At2,Ac2,D1,D2).
copy(N,T,C,D0,D2) :-
          arg(N,T,At),arg(N,C,Ac),
          copy(At,Ac,D0,D1),
          N1 is N-1,copy(N1,T,C,D1,D2).
```

References

[1] F. Benoy and A. King. Inferring argument size relationships with clp(r). In *Proc. of LOPSTR'96*. Springer Verlag, 1996.

[2] M. Carlsson. Boolean constraints in sicstus prolog. Technical Report T91:09, Swedish Institute of Computer Science, 1994.

[3] P. Codognet and G. Filé. Computations, abstractions and contraints in logic programming. *Proc. of ICCL'92*, 1992.

[4] B. De Backer and H. Beringer. A clp language handling disjunctions of linear constraints. In *Proc. of ICLP'93*, pages 550–563. MIT Press, 1993.

[5] R. Giacobazzi, S.K. Debray, and G. Levi. A generalized semantics for constraint logic programs. *Proc. of FGCS'92*, pages 581–591, 1992.

[6] C. Holzbaur. Ofai clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute, 1995.

[7] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *J. Logic Programming*, pages 503–581, 1994.

[8] F. Mesnard. Towards automatic control for clp(χ) programs. *Proc. of LOPSTR'95*, pages 106–119, 1995.

[9] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. *Proc. of JICSLP'96*, pages 7–21, 1996.

[10] F. Mesnard, S. Hoarau, and A. Maillard. Clp(χ) for proving program properties. In *Frontiers of Combining Systems*, pages 321–338. Kluwer Academic Publishers, 1996.