OFAI clp(Q,R) Manual

This Manual documents a Prolog implementation of clp(Q,R), based on SICStus featuring extensible unification via attributed variables.

Edition 1.3.3
December 1995

Christian Holzbaur christian@ai.univie.ac.at

Austrian Research Institute for Artificial Intelligence (OFAI)
Schottengasse 3
A-1010 Vienna, Austria

Constraint Logic Programming over Rationals or Reals

Introduction

The clp(Q,R) system described in this document is an instance of the general Constraint Logic Programming scheme introduced by [Jaffar & Michaylov 87].

The implementation is at least as complete as other existing clp(R) implementations: It solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

The full clp(Q,R) distribution, including a stand-alone manual and an examples directory that is possibly more up to date than the version in the SICStus Prolog distribution, is available from: `http://www.ai.univie.ac.at/clpqr/`.

Referencing this Software

When referring to this implementation of clp(Q,R) in publications, you should use the following reference:

> Holzbaur C.: OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

Acknowledgments

The development of this software was supported by the Austrian *Fonds zur Foerderung der Wissenschaftlichen Forschung* under grant P9426-PHY. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science and Research.

We include a collection of examples that has been distributed with the Monash University version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

Solver Interface

Until rational numbers become first class citizens in SICStus Prolog, rational arithmetics has to be emulated. Because of the emulation it is too expensive to support arithmetics with automatic coercion between all sorts of numbers, like you find it in CommonLisp, for example.

You must choose whether you want to operate in the field of Q (Rationals) or R (Reals):

```
| ?- use_module(library(clpq)).
```

or

```
| ?- use_module(library(clpr)).
```

Notational Conventions

Throughout this chapter, the prompts `clp(q) ?-` and `clp(r) ?-` are used to differentiate between clp(Q) and clp(R) in exemplary interactions.

In general there are many ways to express the same linear relationship. This degree of freedom is manifest in the fact that the printed manual and an actual interaction with the current version of clp(Q,R) may show syntactically different answer constraints, despite the fact the same semantic relationship is being expressed. There are means to control the presentation, see [Variable Ordering], page 12. The approximative nature of floating point numbers may also produce numerical differences between the text in this manual and the actual results of clp(R), for a given edition of the software.

Solver Predicates

The solver interface for both Q and R consists of the following predicates which are exported from `module(linear)`.

`{+`*Constraint*`}`

Constraint is a term accepted by the the grammar below. The corresponding constraint is added to the current constraint store and checked for satisfiability. If you want to overload `{}/1` with other solvers, you can avoid its importation via: `use_module(clpq, [])`.

Constraint	`-->`	*C*	
		`|` *C* `,` *C*	conjunction

C `-->`	*Expr* `=:=` *Expr*	equation
	`|` *Expr* `=` *Expr*	equation
	`|` *Expr* `<` *Expr*	strict inequation
	`|` *Expr* `>` *Expr*	strict inequation
	`|` *Expr* `=<` *Expr*	nonstrict inequation
	`|` *Expr* `>=` *Expr*	nonstrict inequation
	`|` *Expr* `=\=` *Expr*	disequation

Expr `-->`	*variable*	Prolog variable
	`|` *number*	floating point or integer
	`|` `+` *Expr*	unary plus
	`|` `-` *Expr*	unary minus
	`|` *Expr* `+` *Expr*	addition
	`|` *Expr* `-` *Expr*	subtraction
	`|` *Expr* `*` *Expr*	multiplication
	`|` *Expr* `/` *Expr*	division
	`|` *abs(Expr)*	absolute value
	`|` *sin(Expr)*	trigonometric sine
	`|` *cos(Expr)*	trigonometric cosine
	`|` *tan(Expr)*	trigonometric tangent
	`|` *pow(Expr,Expr)*	raise to the power
	`|` *exp(Expr,Expr)*	raise to the power
	`|` *min(Expr,Expr)*	minimum of the two arguments
	`|` *max(Expr,Expr)*	maximum of the two arguments
	`|` *#(Const)*	symbolic numerical constants

Conjunctive constraints `{C,C}` have been made part of the syntax in order to enable grouped submission of constraints, which could be exploited by future versions of this software. Symbolic numerical constants are provided for compatibility only, see [Monash Examples], page 18.

`entailed(+`*Constraint*`)`

Succeeds iff the linear *Constraint* is entailed by the current constraint store. This predicate does not change the state of the constraint store.

```
clp(q) ?- {A =< 4}, entailed(A=\=5).

{A=<4}
yes

clp(q) ?- {A =< 4}, entailed(A=\=3).

no
```

`inf(+`*Expr*`, -`*Inf*`)`

Computes the infimum of the linear expression *Expr* and unifies it with *Inf*. Failure indicates unboundedness.

`sup(+`*Expr*`, -`*Sup*`)`

Computes the supremum of the linear expression *Expr* and unifies it with *Sup*. Failure indicates unboundedness.

```
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11,
            X+3*Y =< 15, Z = 30*X+50*Y
          }, sup(Z, Sup).

Sup = 310,
{Z=30*X+50*Y},
{X+1/2*Y=<8},
```

```
{X+3*Y=<15},
{X+2*Y=<11}
```

`minimize(+`*Expr*`)`

Computes the infimum of the linear expression *Expr* and equates it with the expression, i.e. as if defined as:

```
minimize(Expr) :- inf(Expr, Expr).
```

`maximize(+`*Expr*`)`

Computes the supremum of the linear expression *Expr* and equates it with the expression.

```
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11,
            X+3*Y =< 15, Z = 30*X+50*Y
          }, maximize(Z).

X = 7,
Y = 2,
Z = 310
```

`bb_inf(+`*Ints*`, +`*Expr*`, -`*Inf*`)`

Computes the infimum of the linear expression *Expr* under the additional constraint that all of variables in the list *Ints* assume integral values at the infimum. This allows for the solution of mixed integer linear optimization problems, see [MIP], page 20.

`ordering(+`*Spec*`)`

Provides a means to control one aspect of the presentation of the answer constraints, see [Variable Ordering], page 12.

Unification

Equality constraints are added to the store implicitly each time variables that have been mentioned in explicit constraints are bound - either to another such variable or to a number.

```
clp(r) ?- {2*A+3*B=C/2}, C=10.0, A=B.

A = 1.0,
B = 1.0,
C = 10.0
```

Is equivalent modulo rounding errors to

```
clp(r) ?- {2*A+3*B=C/2, C=10, A=B}.

A = 1.0,
B = 0.9999999999999999,
C = 10.0
```

The shortcut bypassing the use of `{}/1` is allowed and makes sense because the interpretation of this equality in Prolog and clp(R) coincides. In general, equations involving interpreted functors, `+/2` in this case, must be fed to the solver explicitly:

```
clp(r) ?- X=3.0+1.0, X=4.0.

no
```

Further, variables known by clp(R) may be bound directly to floats only. Likewise, variables known by clp(Q) may be bound directly to rational numbers only, see [Rationals], page 22. Failing to do so is rewarded with an exception:

```
clp(q) ?- {2*A+3*B=C/2}, C=10.0, A=B.
{ERROR: not_normalized(10.0)}
```

This is because `10.0` is not a rational constant. To make clp(Q) happy you have to say:

```
clp(q) ?- {2*A+3*B=C/2}, C=rat(10,1), A=B.

A = 1,
B = 1,
C = 10
```

If you use `{}/1`, you don't have to worry about such details. Alternatively, you may use the automatic expansion facility, check [Syntactic Sugar], page 17.

Feedback and Bindings

What was covered so far was how the user populates the constraint store. The other direction of the information flow consists of the success and failure of the above predicates and the binding of variables to numerical values and the aliasing of variables. Example:

```
clp(r) ?- {A-B+C=10, C=5+5}.

B = A,
C = 10.0
```

The linear constraints imply `A=B` and the solver consequently exports this binding to the Prolog world, which is manifest in the fact that the test `A==B` will succeed. More about answer presentation in [Projection], page 11.

Linearity and Nonlinear Residues

The clp(Q,R) system is restricted to deal with linear constraints because the decision algorithms for general nonlinear constraints are prohibitively expensive to run. If you need this functionality badly, you should look into symbolic algebra packages. Although the clp(Q,R) system cannot solve nonlinear constraints, it will collect them faithfully in the hope that through the addition of further (linear) constraints they might get simple enough to solve eventually. If an answer contains nonlinear constraints, you have to be aware of the fact that success is qualified modulo the existence of a solution to the system of residual (nonlinear) constraints:

```
clp(r) ?- {sin(X) = cos(X)}.

nonlin:{sin(X)-cos(X)=0.0}
```

There are indeed infinitely many solutions to this constraint (`X = 0.785398 + n*Pi`), but clp(Q,R) has no direct means to find and represent them.

The systems goes through some lengths to recognize linear expressions as such. The method is based on a normal form for multivariate polynomials. In addition, some simple isolation axioms, that can be used in equality constraints, have been added. The current major limitation of the method is that full polynomial division has not been implemented. Examples:

This is an example where the isolation axioms are sufficient to determine the value of X.

```
clp(r) ?- {sin(cos(X)) = 1/2}.

X = 1.0197267436954502
```

If we change the equation into an inequation, clp(Q,R) gives up:

```
clp(r) ?- {sin(cos(X)) < 1/2}.

nonlin:{sin(cos(X))-0.5<0.0}
```

The following is easy again:

```
clp(r) ?- {sin(X+2+2)/sin(4+X) = Y}.

Y = 1.0
```

And so is this:

```
clp(r) ?- {(X+Y)*(Y+X)/X = Y*Y/X+99}.

{Y=49.5-0.5*X}
```

An ancient symbol manipulation benchmark consists in rising the expression `X+Y+Z+1` to the 15th power:

```
clp(q) ?-  {exp(X+Y+Z+1,15)=0}.
nonlin:{Z^15+Z^14*15+Z^13*105+Z^12*455+Z^11*1365+Z^10*3003+...
        ... polynomial continues for a few pages ...
        =0}
```

Computing its roots is another story.

How Nonlinear Residues are made to disappear

Binding variables that appear in nonlinear residues will reduce the complexity of the nonlinear expressions and eventually results in linear expressions:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.

nonlin:{Y*2-X^2*2+Y*X*2+X*2+1=0}
```

Equating X and Y collapses the expression completely and even determines the values of the two variables:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.

X = -1/4,
Y = -1/4
```

Isolation Axioms

These axioms are used to rewrite equations such that the variable to be solved for is moved to the left hand side and the result of the evaluation of the right hand side can be assigned to the variable. This allows, for example, to use the exponentiation operator for the computation of roots and logarithms, see below.

`A = B * C` Residuates unless B or C is ground or A and B or C are ground.

`A = B / C` Residuates unless C is ground or A and B are ground.

`X = min(Y,Z)`
Residuates unless Y and Z are ground.

`X = max(Y,Z)`
Residuates unless Y and Z are ground.

`X = abs(Y)`
Residuates unless Y is ground.

`X = pow(Y,Z), X = exp(Y,Z)`
Residuates unless any pair of two of the three variables is ground. Example:

```
clp(r) ?- { 12=pow(2,X) }.

X = 3.5849625007211565

clp(r) ?- { 12=pow(X,3.585) }.
```

```
X = 1.9999854993443926

clp(r) ?- { X=pow(2,3.585) }.

X = 12.000311914286545
```

`X = sin(Y)`
Residuates unless X or Y is ground. Example:

```
clp(r) ?- { 1/2 = sin(X) }.

X = 0.5235987755982989
```

`X = cos(Y)`
Residuates unless X or Y is ground.

`X = tan(Y)`
Residuates unless X or Y is ground.

Numerical Precision and Rationals

The fact that you can switch between clp(R) and clp(Q) should solve most of your numerical problems regarding precision. Within clp(Q), floating point constants will be coerced into rational numbers automatically. Transcendental functions will be approximated with rationals. The precision of the approximation is limited by the floating point precision. These two provisions allow you to switch between clp(R) and clp(Q) without having to change your programs.

What is to be kept in mind however is the fact that it may take quite big rationals to accommodate the required precision. High levels of precision are for example required if your linear program is ill-conditioned, i.e., in a full rank system the determinant of the coefficient matrix is close to zero. Another situation that may call for elevated levels of precision is when a linear optimization problem requires exceedingly many pivot steps before the optimum is reached.

If your application approximates irrational numbers, you may be out of space particularly soon. The following program implements N steps of Newton's approximation for the square root function at point 2.

```
%
% from file: library('clpqr/examples/root')
%
root(N, R) :-
  root(N, 1, R).

root(0, S, R) :- !, S=R.
root(N, S, R) :-
  N1 is N-1,
  { S1 = S/2 + 1/S },
  root(N1, S1, R).
```

It is known that this approximation converges quadratically, which means that the number of correct digits in the decimal expansion roughly doubles with each iteration. Therefore the numerator and denominator of the rational approximation have to grow likewise:

```
clp(q) ?- use_module(library('clpqr/examples/root')).
clp(q) ?- root(3,R),print_decimal(R,70).
1.4142156862 7450980392 1568627450 9803921568 6274509803 9215686274
5098039215

R = 577/408

clp(q) ?- root(4,R),print_decimal(R,70).
1.4142135623 7468991062 6295578890 1349101165 5962211574 4044584905
0192000543

R = 665857/470832

clp(q) ?- root(5,R),print_decimal(R,70).
1.4142135623 7309504880 1689623502 5302436149 8192577619 7428498289
4986231958

R = 886731088897/627013566048

clp(q) ?- root(6,R),print_decimal(R,70).
1.4142135623 7309504880 1688724209 6980785696 7187537723 4001561013
1331132652

R = 1572584048032918633353217/1111984844349868137938112

clp(q) ?- root(7,R),print_decimal(R,70).
1.4142135623 7309504880 1688724209 6980785696 7187537694 8073176679
7379907324

R = 4946041176255201878775086487573351061418968498177 /
    3497379255757941172020851852070562919437964212608
```

Iterating for 8 steps produces no further change in the first 70 decimal digits of `sqrt(2)`. After 15 steps the approximating rational number has a numerator and a denominator with 12543 digits each, and the next step runs out of memory.

Another irrational number that is easily computed is e. The following program implements an alternating series for `1/e`, where the absolute value of last term is an upper bound on the error.

```
%
% from file: library('clpqr/examples/root')
%
e(N, E) :-
  { Err =:= exp(10,-(N+2)), Half =:= 1/2 },
  inv_e_series(Half, Half, 3, Err, Inv_E),
  { E =:= 1/Inv_E }.

inv_e_series(Term, S0, _, Err, Sum) :-
  { abs(Term) =< Err }, !,
  S0 = Sum.
```

```
inv_e_series(Term, S0, N, Err, Sum) :-
  N1 is N+1,
  { Term1 =:= -Term/N, S1 =:= Term1+S0 },
  inv_e_series(Term1, S1, N1, Err, Sum).
```

The computation of the rational number E that approximates e up to at least 1000 digits in its decimal expansion requires the evaluation of 450 terms of the series, i.e. 450 calls of `inv_e_series/5`.

```
clp(q) ?- e(1000,E).

E = 7149056228932760213666809592072842334290744221392610955845565494
    3708750229467761730471738895197792271346693089326102132000338192
    0131874187833985420922688804220167840319199699494193852403223700
    5853832741544191628747052136402176941963825543565900589161585723
    4023097417605004829991929283045372355639145644588174733401360176
    9953973706537274133283614740902771561159913069917833820285608440
    3104966899999651928637634656418969027076699082888742481392304807
    9484725489080844360397606199771786024695620205344042765860581379
    3538290451208322129898069978107971226873160872046731879753034549
    3130492167474809196348846916421782850086985668680640425192038155
    4902863298351349469211627292865440876581064873866786120098602898
    8799130098877372097360065934827751120659213470528793143805903554
    7928682131082164366007016698761961066948371407368962539467994627
    1374858249110795976398595034606994740186040425117101588480000000
    0000000000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000
    /
    2629990810403002651095959155503002285441272170673105334466808931
    6863103901346024240326549035084528682487048064823380723787110941
    6809235187356318780972302796570251102928552003708556939314795678
    1978390674393498540663747334079841518303636625888963910391440709
    0887345797303470959207883316838346973393937778363411195624313553
    8835644822353659840936818391050630360633734935381528275392050975
    7271468992840907541350345459011192466892177866882264242860412188
    0652112744642450404625763019639086944558899249788084559753723892
    1643188991444945360726899532023542969572584363761073528841147012
    2634218045463494055807073778490814692996517359952229262198396182
    1838930043528583109973872348193806830382584040536394640895148751
    0766256738740729894909630785260101721285704616818889741995949666
    6303289703199393801976334974240815397920213059799071915067856758
    6716458821062645562512745336709063396510021681900076680696945309
    3660590933279867736747926648678738515702777431353845466199680991
    733618734211521654777749116601082000 59
```

The decimal expansion itself looks like this:

```
clp(q) ?- e(1000, E), print_decimal(E, 1000).
2.
7182818284 5904523536 0287471352 6624977572 4709369995 9574966967
6277240766 3035354759 4571382178 5251664274 2746639193 2003059921
8174135966 2904357290 0334295260 5956307381 3232862794 3490763233
8298807531 9525101901 1573834187 9307021540 8914993488 4167509244
7614606680 8226480016 8477411853 7423454424 3710753907 7744992069
5517027618 3860626133 1384583000 7520449338 2656029760 6737113200
```

```
7093287091 2744374704 7230696977 2093101416 9283681902 5515108657
4637721112 5238978442 5056953696 7707854499 6996794686 4454905987
9316368892 3009879312 7736178215 4249992295 7635148220 8269895193
6680331825 2886939849 6465105820 9392398294 8879332036 2509443117
3012381970 6841614039 7019837679 3206832823 7646480429 5311802328
7825098194 5581530175 6717361332 0698112509 9618188159 3041690351
5988885193 4580727386 6738589422 8792284998 9208680582 5749279610
4841984443 6346324496 8487560233 6248270419 7862320900 2160990235
3043699418 4914631409 3431738143 6405462531 5209618369 0888707016
7683964243 7814059271 4563549061 3031072085 1038375051 0115747704
1718986106 8739696552 1267154688 9570350354
```

Projection and Redundancy Elimination

Once a derivation succeeds, the Prolog system presents the bindings for the variables in the query. In a CLP system, the set of answer constraints is presented in analogy. A complication in the CLP context are variables and associated constraints that were not mentioned in the query. A motivating example is the familiar `mortgage` relation:

```
%
% from file: library('clpqr/examples/mg')
%
mg(P,T,I,B,MP):-
  {
     T = 1,
     B + MP = P * (1 + I)
  }.
mg(P,T,I,B,MP):-
  {
     T > 1,
     P1 = P * (1 + I) - MP,
     T1 = T - 1
  },
  mg(P1, T1, I, B, MP).
```

A sample query yields:

```
clp(r) ?- use_module(library('clpqr/examples/mg')).
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

Without projection of the answer constraints onto the query variables we would observe the following interaction:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=12.682503013196973*_A-11.682503013196971*P},
{Mp= -(_A)+1.01*P},
{_B=2.01*_A-1.01*P},
```

```
{_C=3.0301*_A-2.0301*P},
{_D=4.060401000000001*_A-3.0604009999999997*P},
{_E=5.101005010000001*_A-4.10100501*P},
{_F=6.152015060100001*_A-5.152015060099999*P},
{_G=7.213535210701001*_A-6.213535210700999*P},
{_H=8.285670562808011*_A-7.285670562808009*P},
{_I=9.368527268436091*_A-8.36852726843609*P},
{_J=10.462212541120453*_A-9.46221254112045*P},
{_K=11.566834666531657*_A-10.566834666531655*P}
```

The variables *_A* ... *_K* are not part of the query, they originate from the mortgage program proper. Although the latter answer is equivalent to the former in terms of linear algebra, most users would prefer the former.

Variable Ordering

In general, there are many ways to express the same linear relationship between variables. clp(Q,R) does not care to distinguish between them, but the user might. The predicate `ordering(+`*Spec*`)` gives you some control over the variable ordering. Suppose that instead of *B*, you want *Mp* to be the defined variable:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

This is achieved with:

```
clp(r) ?-  mg(P,12,0.01,B,Mp), ordering([Mp]).

{Mp= -0.0788487886783417*B+0.08884878867834171*P}
```

One could go one step further and require *P* to appear before (to the left of) *B* in a addition:

```
clp(r) ?- mg(P,12,0.01,B,Mp), ordering([Mp,P]).

{Mp=0.08884878867834171*P-0.0788487886783417*B}
```

Spec in `ordering(+`*Spec*`)` is either a list of variables with the intended ordering, or of the form *A*`<`*B*. The latter form means that *A* goes to the left of *B*. In fact, `ordering([A,B,C,D])` is shorthand for:

```
ordering(A < B), ordering(A < C), ordering(A < D),
ordering(B < C), ordering(B < D),
ordering(C < D)
```

The ordering specification only affects the final presentation of the constraints. For all other operations of clp(Q,R), the ordering is immaterial. Note that `ordering/1` acts like a constraint: you can put it anywhere in the computation, and you can submit multiple specifications.

```
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp).

{B= -12.682503013196973*Mp+1.1268250301319698*P}

yes
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp), ordering(P < Mp).

{P=0.8874492252651537*B+11.255077473484631*Mp}
```

Turning Answers into Terms

In meta-programming applications one needs to get a grip on the results computed by the clp(Q,R) solver. The SISCtus Prolog predicate `call_residue/2` provides this functionality:

```
clp(r) ?- call_residue({2*A+B+C=10,C-D=E,A<10}, Constraints).

Constraints = [
                [A]-{A<10.0},
                [B]-{B=10.0-2.0*A-C},
                [D]-{D=C-E}
              ]
```

Projecting Inequalities

As soon as linear inequations are involved, projection gets more demanding complexity wise. The current clp(Q,R) version uses a Fourier-Motzkin algorithm for the projection of linear inequalities. The choice of a suitable algorithm is somewhat dependent on the number of variables to be eliminated, the total number of variables, and other factors. It is quite easy to produce problems of moderate size where the elimination step takes some time. For example, when the dimension of the projection is 1, you might be better off computing the supremum and the infimum of the remaining variable instead of eliminating `n-1` variables via implicit projection.

In order to make answers as concise as possible, redundant constraints are removed by the system as well. In the following set of inequalities, half of them are redundant.

```
%
% from file: library('clpqr/examples/elimination')
%
```

```
example(2, [X0,X1,X2,X3,X4]) :-
  {
       +87*X0  +52*X1  +27*X2  -54*X3  +56*X4 =<  -93,
       +33*X0  -10*X1  +61*X2  -28*X3  -29*X4 =<   63,
       -68*X0   +8*X1  +35*X2  +68*X3  +35*X4 =<  -85,
       +90*X0  +60*X1  -76*X2  -53*X3  +24*X4 =<  -68,
       -95*X0  -10*X1  +64*X2  +76*X3  -24*X4 =<   33,
       +43*X0  -22*X1  +67*X2  -68*X3  -92*X4 =<  -97,
       +39*X0   +7*X1  +62*X2  +54*X3  -26*X4 =<  -27,
       +48*X0  -13*X1   +7*X2  -61*X3  -59*X4 =<   -2,
       +49*X0  -23*X1  -31*X2  -76*X3  +27*X4 =<    3,
       -50*X0  +58*X1   -1*X2  +57*X3  +20*X4 =<    6,
       -13*X0  -63*X1  +81*X2   -3*X3  +70*X4 =<   64,
       +20*X0  +67*X1  -23*X2  -41*X3  -66*X4 =<   52,
       -81*X0  -44*X1  +19*X2  -22*X3  -73*X4 =<  -17,
       -43*X0   -9*X1  +14*X2  +27*X3  +40*X4 =<   39,
       +16*X0  +83*X1  +89*X2  +25*X3  +55*X4 =<   36,
        +2*X0  +40*X1  +65*X2  +59*X3  -32*X4 =<   13,
       -65*X0  -11*X1  +10*X2  -13*X3  +91*X4 =<   49,
       +93*X0  -73*X1  +91*X2   -1*X3  +23*X4 =<  -87
  }.
```

Consequently, the answer consists of the system of nine non-redundant inequalities only:

```
clp(q) ?- use_module(library('clpqr/examples/elimination')).
clp(q) ?- example(2, [X0,X1,X2,X3,X4]).

{X0-2/17*X1-35/68*X2-X3-35/68*X4>=5/4},
{X0-73/93*X1+91/93*X2-1/93*X3+23/93*X4=<-29/31},
{X0-29/25*X1+1/50*X2-57/50*X3-2/5*X4>=-3/25},
{X0+7/39*X1+62/39*X2+18/13*X3-2/3*X4=<-9/13},
{X0+2/19*X1-64/95*X2-4/5*X3+24/95*X4>=-33/95},
{X0+2/3*X1-38/45*X2-53/90*X3+4/15*X4=<-34/45},
{X0-23/49*X1-31/49*X2-76/49*X3+27/49*X4=<3/49},
{X0+44/81*X1-19/81*X2+22/81*X3+73/81*X4>=17/81},
{X0+9/43*X1-14/43*X2-27/43*X3-40/43*X4>=-39/43}
```

The projection (the shadow) of this polyhedral set into the `X0,X1` space can be computed via the implicit elimination of non-query variables:

```
clp(q) ?- example(2, [X0,X1|_]).

{X0+2619277/17854273*X1>=-851123/17854273},
{X0+6429953/16575801*X1=<-12749681/16575801},
{X0+19130/1213083*X1>=795400/404361},
{X0-1251619/3956679*X1>=21101146/3956679},
{X0+601502/4257189*X1>=220850/473021}
```

Projection is quite a powerful concept that leads to surprisingly terse executable specifications of nontrivial problems like the computation of the convex hull from a set of points in an n-dimensional space: Given the program

```
%
```

```
% from file: library('clpqr/examples/elimination')
%
conv_hull(Points, Xs) :-
  lin_comb(Points, Lambdas, Zero, Xs),
  zero(Zero),
  polytope(Lambdas).

polytope(Xs) :-
  positive_sum(Xs, 1).

  positive_sum([], Z) :- {Z=0}.
  positive_sum([X|Xs], SumX) :-
    { X >= 0, SumX = X+Sum },
    positive_sum(Xs, Sum).

zero([]).
zero([Z|Zs]) :- {Z=0}, zero(Zs).

lin_comb([],         [],     S1, S1).
lin_comb([Ps|Rest], [K|Ks], S1, S3) :-
  lin_comb_r(Ps, K, S1, S2),
  lin_comb(Rest, Ks, S2, S3).

  lin_comb_r([],      _, [],     []).
  lin_comb_r([P|Ps], K, [S|Ss], [Kps|Ss1]) :-
    { Kps = K*P+S },
    lin_comb_r(Ps, K, Ss, Ss1).
```

we can post the following query:

```
clp(q) ?- conv_hull([ [1,1], [2,0], [3,0], [1,2], [2,2] ], [X,Y]).

{Y=<2},
{X+1/2*Y=<3},
{X>=1},
{Y>=0},
{X+Y>=2}
```

This answer is easily verified graphically:

```
   |
 2 -    *    *
   |
   |
 1 -    *
   |
   |
 0 -----|----*----*----
        1    2    3
```

The convex hull program directly corresponds to the mathematical definition of the convex hull. What does the trick in operational terms is the implicit elimination of the *Lambdas* from the program formulation. Please note that this program does not limit the number of points or the dimension of the space they are from. Please note further that quantifier elimination is a computationally expensive operation and therefore this program is only useful as a benchmark for the projector and not so for the intended purpose.

Why Disequations

A beautiful example of disequations at work is due to [Colmerauer 90]. It addresses the task of tiling a rectangle with squares of all-different, a priori unknown sizes. Here is a translation of the original `Prolog-III` program to clp(Q,R):

```
%
% from file: library('clpqr/examples/squares')
%
filled_rectangle(A, C) :-
  { A >= 1 },
  distinct_squares(C),
  filled_zone([-1,A,1], _, C, []).

distinct_squares([]).
distinct_squares([B|C]) :-
  { B > 0 },
  outof(C, B),
  distinct_squares(C).

outof([],     _).
outof([B1|C], B) :-
  { B =\= B1 },        % *** note disequation ***
  outof(C, B).

filled_zone([V|L], [W|L], C0, C0) :-
  { V=W,V >= 0 }.
filled_zone([V|L], L3, [B|C], C2) :-
  { V < 0 },
  placed_square(B, L, L1),
  filled_zone(L1, L2, C, C1),
  { Vb=V+B },
  filled_zone([Vb,B|L2], L3, C1, C2).

placed_square(B, [H,H0,H1|L], L1) :-
  { B > H, H0=0, H2=H+H1 },
  placed_square(B, [H2|L], L1).
placed_square(B, [B,V|L], [X|L]) :-
  { X=V-B }.
placed_square(B, [H|L], [X,Y|L]) :-
  { B < H, X= -B, Y=H-B }.
```

There are no tilings with less than nine squares except the trivial one where the rectangle equals the only square. There are eight solutions for nine squares. Six further solutions are rotations of the first two.

```
clp(q) ?- use_module(library('clpqr/examples/squares')).
clp(q) ?- filled_rectangle(A, Squares).

A = 1,
Squares = [1] ? ;

A = 33/32,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]
```

Depending on your hardware, the above query may take a few minutes. Supplying the knowledge about the minimal number of squares beforehand cuts the computation time by a factor of roughly four:

```
clp(q) ?- length(Squares, 9), filled_rectangle(A, Squares).

A = 33/32,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]
```

Syntactic Sugar

There is a package that transforms programs and queries from a eval-quote variant of clp(Q,R) into corresponding programs and queries in a quote-eval variant. Before you use it, you need to know that in an eval-quote language, all symbols are interpreted unless explicitly quoted. This means that interpreted terms cannot be manipulated syntactically directly. Meta-programming in a CLP context by definition manipulates interpreted terms, therefore you need `quote/1` (just as in LISP) and some means to put syntactical terms back to their interpreted life: `{}/1`.

In a quote-eval language, meta-programming is (pragmatically) simpler because everything is implicitly quoted until explicitly evaluated. On the other hand, now object programming suffers from the dual inconvenience.

We chose to make our version of clp(Q,R) of the quote-eval type because this matches the intended use of the already existing boolean solver of SICStus. In order to keep the users of the eval-quote variant happy, we provide a source transformation package. It is activated via:

```
| ?- use_module(library('clpqr/expand')).
```

Loading the package puts you in a mode where the arithmetic functors like `+/2`, `*/2` and all numbers (functors of arity 0) are interpreted semantically.

```
clp(r) ?- 2+2=X.

X = 4.0
```

The package works by *purifying* programs and queries in the sense that all references to interpreted terms are made explicit. The above query is expanded prior to evaluation into:

```
linear:{2.0+2.0=X}
```

The same mechanism applies when interpreted terms are nested deeper:

```
some_predicate(10, f(A+B/2), 2*cos(A))
```

Expands into:

```
linear:{Xc=2.0*cos(A)},
linear:{Xb=A+B/2},
linear:{Xa=10.0},
some_predicate(Xa, f(Xb), Xc)
```

This process also applies when files are consulted or compiled. In fact, this is the only situation where expansion can be applied with relative safety. To see this, consider what happens when the toplevel evaluates the expansion, namely some calls to the clp(Q,R) solver, followed by the call of the purified query. As we learned in [Feedback], page 5, the solver may bind variables, which produces a goal with interpreted functors in it (numbers), which leads to another stage of expansion, and so on.

We recommend that you only turn on expansion temporarily while consulting or compiling files needing expansion with `expand/0` and `noexpand/0`.

Monash Examples

This collection of examples has been distributed with the Monash University Version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

In order to execute the examples, a small compatibility package has to be loaded first:

```
clp(r) ?- use_module(library('clpqr/monash')).
```

Then, assuming you are using clp(R):

```
clp(r) ?- expand, [library('clpqr/examples/monash/rkf45')],
          noexpand.

clp(r) ?- go.
Point    0.00000 :    0.75000    0.00000
Point    0.50000 :    0.61969    0.47793
Point    1.00000 :    0.29417    0.81233
Point    1.50000 :   -0.10556    0.95809
Point    2.00000 :   -0.49076    0.93977
Point    2.50000 :   -0.81440    0.79929
Point    3.00000 :   -1.05440    0.57522

Iteration finished
------------------
 439  derivative evaluations
```

Compatibility Notes

The Monash examples have been written for clp(R). Nevertheless, all but `rkf45` complete nicely in clp(Q). With `rkf45`, clp(Q) runs out of memory. This is an instance of the problem discussed in [Numerical Precision], page 8.

The Monash University clp(R) interpreter features a `dump/n` predicate. It is used to print the target variables according to the given ordering. Within this version of clp(Q,R), the corresponding functionality is provided via `ordering/1`. The difference is that `ordering/1` does only specify the ordering of the variables and *no* printing is performed. We think Prolog has enough predicates to perform output already. You can still run the examples referring to `dump/n` from the Prolog toplevel:

```
clp(r) ?- expand, [library('clpqr/examples/monash/mortgage')], noexpand.

% go2
%
clp(r) ?- mg(P,120,0.01,0,MP), dump([P,MP]).

{P=69.7005220313972*MP}

% go3
%
clp(r) ?- mg(P,120,0.01,B,MP), dump([P,B,MP]).

{P=0.3029947796860270б*B+69.7005220313972*MP}

% go4
%
clp(r) ?- mg(999, 3, Int, 0, 400), dump.

nonlin:{_B-_B*Int+_A+400.0=0.0},
nonlin:{_A-_A*Int+400.0=0.0},
```

```
{_B=599.0+999.0*Int}
```

A Mixed Integer Linear Optimization Example

In this section we are going to exercise our solver a little by the computation of a small mixed integer optimization problem (MIP) from miplib, a collection of MIP models, housed at Rice University. Here are the original comments on the example:

```
NAME:          flugpl
ROWS:          18
COLUMNS:       18
INTEGER:       11
NONZERO:       46
BEST SOLN:     1201500 (opt)
LP SOLN:       1167185.73
SOURCE:        Harvey M. Wagner
               John W. Gregory (Cray Research)
               E. Andrew Boyd (Rice University)
APPLICATION:   airline model
COMMENTS:      no integer variables are binary
```

```
%
% from file: library('clpqr/examples/mip')
%
example(flugpl, Obj, Vs, Ints, []) :-
  Vs = [ Anm1,Anm2,Anm3,Anm4,Anm5,Anm6,
         Stm1,Stm2,Stm3,Stm4,Stm5,Stm6,
         UE1,UE2,UE3,UE4,UE5,UE6],
  Ints = [Stm6, Stm5, Stm4, Stm3, Stm2,
          Anm6, Anm5, Anm4, Anm3, Anm2, Anm1],

  Obj =    2700*Stm1 + 1500*Anm1 + 30*UE1
         + 2700*Stm2 + 1500*Anm2 + 30*UE2
         + 2700*Stm3 + 1500*Anm3 + 30*UE3
         + 2700*Stm4 + 1500*Anm4 + 30*UE4
         + 2700*Stm5 + 1500*Anm5 + 30*UE5
         + 2700*Stm6 + 1500*Anm6 + 30*UE6,
```

```
allpos(Vs),
{  Stm1 = 60, 0.9*Stm1 +1*Anm1 -1*Stm2 = 0,
   0.9*Stm2 +1*Anm2 -1*Stm3 = 0, 0.9*Stm3 +1*Anm3 -1*Stm4 = 0,
   0.9*Stm4 +1*Anm4 -1*Stm5 = 0, 0.9*Stm5 +1*Anm5 -1*Stm6 = 0,
   150*Stm1 -100*Anm1 +1*UE1 >= 8000,
   150*Stm2 -100*Anm2 +1*UE2 >= 9000,
   150*Stm3 -100*Anm3 +1*UE3 >= 8000,
   150*Stm4 -100*Anm4 +1*UE4 >= 10000,
   150*Stm5 -100*Anm5 +1*UE5 >= 9000,
   150*Stm6 -100*Anm6 +1*UE6 >= 12000,
   -20*Stm1 +1*UE1 =< 0, -20*Stm2 +1*UE2 =< 0, -20*Stm3 +1*UE3 =< 0,
   -20*Stm4 +1*UE4 =< 0, -20*Stm5 +1*UE5 =< 0, -20*Stm6 +1*UE6 =< 0,
   Anm1 =< 18, 57 =< Stm2, Stm2 =< 75, Anm2 =< 18,
   57 =< Stm3, Stm3 =< 75, Anm3 =< 18, 57 =< Stm4,
   Stm4 =< 75, Anm4 =< 18, 57 =< Stm5, Stm5 =< 75,
   Anm5 =< 18, 57 =< Stm6, Stm6 =< 75, Anm6 =< 18
 }.

allpos([]).
allpos([X|Xs]) :- {X >= 0}, allpos(Xs).
```

We can first check whether the relaxed problem has indeed the quoted infimum:

```
clp(r) ?- example(flugpl, Obj, _, _, _), inf(Obj, Inf).

Inf = 1167185.7255923203
```

Computing the infimum under the additional constraints that `Stm6`, `Stm5`, `Stm4`, `Stm3`, `Stm2`, `Anm6`, `Anm5`, `Anm4`, `Anm3`, `Anm2`, `Anm1` assume integer values at the infimum is computationally harder, but the query does not change much:

```
clp(r) ?- example(flugpl, Obj, _, Ints, _), bb_inf(Ints, Obj, Inf).

Inf = 1201500.0000000005
```

Implementation Architecture

The system consists roughly of the following components:

- A polynomial normal form expression simplification mechanism.
- A solver for linear equations [Holzbaur 92].
- A simplex algorithm to decide linear inequalities [Holzbaur 94].

Fragments and Bits

Rationals

The internal data structure for rational numbers is `rat(Num,Den)`. *Den* is always positive, i.e. the sign of the rational number is the sign of *Num*. Further, *Num* and *Den* are relative prime. Note that integer *N* looks like `rat(N,1)` in this representation. You can control printing of terms with `portray/1`.

Partial Evaluation, Compilation

Once one has a working solver, it is obvious and attractive to run the constraints in a clause definition at read time or compile time and proceed with the answer constraints in place of the original constraints. This gets you constant folding and in fact the full algebraic power of the solver applied to the avoidance of computations at runtime. The mechanism to realize this idea is to use `call_residue/2` for the expansion of `{}/1` (see ⟨undefined⟩ [Definite], page ⟨undefined⟩, hook predicate `user:goal_expansion/3`).

Asserting with Constraints

If you use the dynamic data base, the clauses you assert might have constraints on the variables occurring in the clause. This works as expected:

```
clp(r) ?- {A < 10}, assert(p(A)).

{A<10.0}

yes
clp(r) ?- p(X).

{X<10.0}
```

Bugs

- The fuzzy comparison of floats is the source for all sorts of weirdness. If a result in R surprises you, try to run the program in Q before you send me a bug report.
- The projector for floundered nonlinear relations keeps too many variables. Its output is rather unreadable.
- Disequations are not projected properly.
- This list is probably incomplete.

Please send bug reports to `<christian@ai.univie.ac.at>`.

References

[Colmerauer 90]
Colmerauer A.: An Introduction to Prolog III, Communications of the ACM, 33(7), 69-90, 1990.

[Heintze et al. 87]
Heintze N., Jaffar J., Michaylov S., Stuckey P., Yap R.: The CLP(R) Programmers Manual, Monash University, Clayton, Victoria, Australia, Department of Computer Science, 1987.

[Holzbaur 92]
Holzbaur C.: A High-Level Approach to the Realization of CLP Languages, in Proceedings of the JICSLP92 Post-Conference Workshop on Constraint Logic Programming Systems, Washington D.C., 1992.

[Holzbaur 92]
Holzbaur C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification, in Bruynooghe M. & Wirsing M.(eds.), Programming Language Implementation and Logic Programming, Springer, LNCS 631, pp.260- 268, 1992.

[Holzbaur 94]
Holzbaur C.: A Specialized, Incremental Solved Form Algorithm for Systems of Linear Inequalities, Austrian Research Institute for Artificial Intelligence, Vienna, TR-94-07, 1994.

[Jaffar & Michaylov 87]
Jaffar J., Michaylov S.: Methodology and Implementation of a CLP System, in Lassez J.L.(ed.), Logic Programming - Proceedings of the 4th International Conference - Volume 1, MIT Press, Cambridge, MA, 1987.

Index of Predicates