Defeasibility in CLP($\mathcal{Q}$) through Generalized Slack Variables

Christian Holzbaur[†], Francisco Menezes[‡] and Pedro Barahona[‡]

[†]Austrian Research Institute for Artificial Intelligence, and
Department of Medical Cybernetics and Artificial Intelligence
University of Vienna
Freyung 6, A-1010 Vienna, Austria
email: christian@ai.univie.ac.at

[‡]Departamento de Informática
Faculdade de Ciências e Técnologia
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
email: {fm,pb}@fct.unl.pt

Abstract. This paper presents a defeasible constraint solver for the domain of linear equations, disequations and inequalities over the body of rational/real numbers. As extra requirements resulting from the incorporation of the solver into an Incremental Hierarchical Constraint Solver (IHCS) scenario we identified: a)the ability to refer to individual constraints by a label, b) the ability to report the (minimal) cause for the unsatisfiability of a set of constraints, and c) the ability to undo the effects of a formerly activated constraint.
We develop the new functionalities after starting the presentation with a general architecture for defeasible constraint solving, through a solved form algorithm that utilizes a generalized, incremental variant of the Simplex algorithm, where the domain of a variable can be restricted to an arbitrary interval. We demonstrate how generalized slacks form the basis for the computation of explanations regarding the cause of unsatisfiability and/or entailment in terms of the constraints told, and the possible deactivation of constraints as demanded by the hierarchy handler.

Keywords: Constraint Logic Programming, Linear Programming, Defeasible Constraint Solving

1 Introduction

Although Constraint Logic Programming enhances the limited expressive power and execution efficiency of Logic Programming, it is insufficient to cope with problems for which many solutions might satisfy a set of mandatory (or hard) constraints of the problem, but where some solutions are preferred to others. In this case, the user should somehow select from the set of all solutions found

by a CLP program those that (s)he prefers, and this is not practical when the solution set is large.

An alternative approach is to use an overconstrained specification, including both hard constraints and soft constraints (that merely specify preferences), and have a system to compute the solutions that satisfy in the best possible way a subset of these preference constraints. This was the approach taken by [2, 16], that proposed an HCLP scheme that allows non-required (or soft) constraints to be specified with some preference level and rely on a constraint solver that explores this hierarchy of constraints to detect the best solutions.

Although the scheme is quite general, little details were published on the implementation of this scheme. In [13, 14] we presented, IHCS, an efficient and incremental defeasible constraint solver that is used as the kernel of an HCLP instance for finite domains. The key points of our implementation were a) early detection of failures through the use of the usual node- and arc-consistency techniques for these domains; b) detection of conflict sets, i.e. the sets of constraints responsible for the failures (this is done by keeping dependencies between constraints through shared variables by adaptation of the AC-5 algorithm [12]); c) selection from these conflict sets of constraints that should be relaxed, together with the selection of constraints (currently relaxed because of conflicts with the former) that can now be safely reactivated; and d) defeating the constraints, i.e. remove the effects of relaxed constraints avoiding reevaluation from scratch.

Although our scheme could in principle be applied to any other domain, early implementation of IHCS did not separate the constraint solver (responsible for the detection of failures and their causes) from the hierarchy manager (responsible for choosing, given some preference criterion, which constraints to relax and which to reactivate).

Moreover, the constraint solver detected unsatisfiable sets of constraints by means of constraint propagation on a constraint network, and the method is not applicable to domains that do not use this representation of constraints. This is the case with (linear) constraint solvers over the reals/rationals which rely on algebraic methods (e.g. some variant of the Simplex algorithm). To effectively apply our scheme to other domains it was thus necessary a) to clearly separate the constraint solver component from the hierarchy handler, and b) to enhance constraint solvers of these domains to cope with the new demands of defeasibility.

These requirements are twofold. On the one hand, the constraint solver must be able to explain the cause of unsatisfiability and/or entailment in terms of the constraints told. On the other hand, it must be able to cope with the incremental activation and deactivation of constraints as demanded by the hierarchy handler.

In this paper we propose a solution to these extensions for CLP($\mathcal{Q}$), a linear constraint solver over the body of rational numbers. Interestingly, both extensions can be realized with the single, simple idea of generalized slack variables.

The following sections will describe a general architecture for defeasible constraint solving, recapture the working of the traditional Simplex algorithm, introduce a variant through the generalization of slack variables, cover the identification of minimal conflict sets, and derive defeasibility.

2 An Architecture for Defeasible Constraint Solving

In [13, 14] an Incremental Hierarchical Constraint Solver (IHCS(X, $\preceq$)) is presented as a general framework to handle, incrementally, hierarchies of constraints in some domain X using some comparator $\preceq$. In these papers, only the instantiation of X to Finite Domains is addressed, and there is no clear division between the component that handles the constraint hierarchies, the comparators used to rank solutions, and the constraint solvers for specific domains.

This section presents the new architecture of IHCS that takes into account such separation, and makes it truly general and able to include different constraint solvers (and thus different domains). Figure 1 depicts this general architecture and the interface among the separate components.

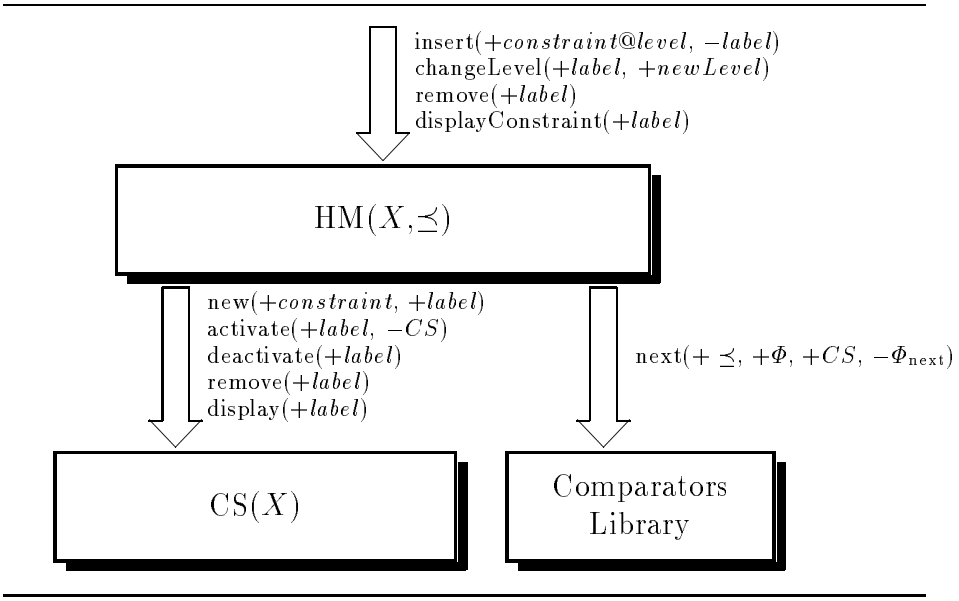

Fig. 1. The Architecture of IHCS(X,$\preceq$)

The functionality IHCS offers to any system requiring defeasible constraint handling (in our case, IHCS is embedded in a Prolog like engine to yield a HCLP(X, $\preceq$) language) is displayed at the top of the Figure. This interface highlights the defeasible nature of IHCS, by including primitives to add or remove a constraint and to promote or demote some existing constraint (by changing its strength or hierarchical level).

The Hierarchy Manager (HM) is responsible for demanding the activation and relaxation of constraints, so as to maintain the best solution. Since it has no specific knowledge of the domain theory X, it must rely on some specialized constraint solver CS(X) to check satisfiability of constraints on domain X. The

insertion of a new constraint into CS(X) is made with predicate new(*constraint*, *label*) which simply creates the constraint with a given label but does not activate it. Constraint labels are required for future reference to the corresponding constraints. This is the case of their activation, via predicate activate(*label*, CS), where CS (the *conflict set*) returns the set of constraints responsible for the possible unsatisfiability of the active constraints (CS is of course empty if these constraints are satisfiable). The reason why constraints are created and activated with different interface entries is that IHCS may require several activations or deactivations (via predicate deactivate(*label*)) of a certain constraint during the search for optimal solutions. A deactivated constraint may be removed from CS(X) with predicate remove(*label*).

A library of comparators includes a set of procedures to compute the next best configuration according to a diversity of criteria. Given the sets of constraints currently active and relaxed (the current configuration Φ), and a conflict set CS (returned by the CS(X) component), predicate next($\preceq$, Φ, CS, Φ_{next}) computes the next configuration Φ_{next} to be tried according to comparator $\preceq$.

To summarize, and to comply with IHCS requirements, a CS(X) must be:

1. **Incremental** - upon the activation of a constraint (demanded through predicate activate(*label*, CS)), the constraint solver must check the satisfiability of the active set of constraints together with the new constraint;
2. **Explanatory** - once unsatisfiability is detected, its causes should be reported to the HM (as a conflict set CS);
3. **Defeasible** - upon the deactivation of a constraint (demanded through predicate deactivate(*label*)), the effects of this formerly activated constraint should be removed avoiding reevaluation from scratch.

Of course all the above requirements impose that, the labels used by the hierarchical manager to refer to individual constraints are shared by the constraint solver.

This requirements are met by our finite domain constraint solver described in [14]. The rest of this paper explains the changes made to a Constraint Solver for linear constraints over rational/real numbers, namely its explanatory and defeasibility enhancements.

3 Simplex with Generalized Slack Variables

Classical Simplex [5] deals with a single sort of slack variables: $S_i \geq 0$. Free variables, negative variables and strict inequalities create minor problems, some of which are addressed by using pairs of slacks.

We will now generalize the concept of slack variables by allowing for arbitrary intervals as the domains of variables. This covers of course the classical Simplex slacks with a non-strict lower bound of zero.

Example 1.

Input constraints	Equations with classical slacks
$X \leq 10$	$X + S_1 = 10$
$X \leq 8$	$X + S_2 = 8$
$X \geq 2$	$-X + S_3 = -2$

Instead of introducing three slack variables with their corresponding rows in the Simplex tableaux, the bounds are represented as attributes of the affected variable directly.

Example 2.

Input constraints	Representation with generalized slacks
$X \leq 10$	
$X \leq 8$	$X_{[2,8]}$
$X \geq 2$	

The next section deals with the interaction of generalized slacks with higher dimensional constraints.

3.1 Solved Form for Inequalities over Bounded Variables

The idea proper has been realized a long time ago in the area of linear programming under the name of *bounded variable linear programs* [15]. In bounded variable linear programs, some or all variables are restricted to lie within individual lower and upper bounds. Such problems can of course be solved by including all bound restrictions as constraints, i.e. rows in the simplex tableau. The advantage of keeping them out of the tableau is that the size of the working basis is smaller. Trivial non-satisfiability, redundancy and implicit equalities are detected by trivial tests of $O(1)$ complexity. Obviously the thread matches and advances current activities in the CLP area that try to restrict the use of general decision methods to the cases where they are unavoidable [10].

We formalize bounded variable linear programs as:

$$\begin{array}{rl} & Minimize\ cx \\ subject\ to & (1.a)\ Ax = b \\ & (1.b)\ l_j \leq x_j \leq u_j \quad for\ j \epsilon J \\ & (1.c)\ x_i\ unrestricted\ for\ i \notin J \end{array} \tag{1}$$

Where $Ax = b$ denotes the subset of the constraints $\{c_i | dim(c_i) > 1\}$, and inequalities have been transformed into equations through the introduction of generalized slack variables. A feasible solution $\mathbf{x}$ of (1) is a *Basic Feasible Solution (BFS)* iff the set

$$\{A_{,j} : j \epsilon J, l_j \leq x_j \leq u_j\} \cup \{A_{,j} : j \notin J\} \tag{2}$$

where $A_{,j}$ is the j-th column vector of A, is linearly independent. A working basis for (1) is a square, nonsingular sub matrix of A of order m. Variables associated

with column vectors of the working basis will be called *basic* variables. All other variables will be called *non-basic* variables. It is clear that a feasible solution $\mathbf{x}$ is an extreme point in the solution space, iff there exists a corresponding working basis with the property that:

1. all non-basic variables are either *at* their lower or upper bound
2. the basic variables are *within* their bounds

The working basis, together with conditions 1 and 2 from above, constitutes our proposed *solved form* for linear inequalities over bounded variables.

Example 3.

Input constraints	Solved form
$x1 + x2 + 2x3 \leq 4,$	$x1_{[0,2]} = 1 + \frac{1}{2}x2 + \frac{1}{2}s2 - s1$ } *basis*
$3x2 + 4x3 \leq 6,$	$x3_{[0,\_]} = \frac{3}{2} - \frac{3}{4}x2 - \frac{1}{4}s2$ } *basis*
$0 \leq x1, x1 \leq 2,$	$s1_{[\overline{0},\_]}$
$0 \leq x2, x2 \leq 9,$	$s2_{[\overline{0},\_]}$
$0 \leq x3$	$x2_{[\overline{0},9]}$

Notational conventions: $x1_{[0,2]}$ means that $x1$ has a (non-strict) lower bound of zero and a (non-strict) upper bound of two. An unspecified bound is denoted as in $x3_{[0,\_]}$, where we have no finite upper bound. The active bound of non-basic variables is denoted by overlining as in $x2_{[\overline{0},9]}$. If you insert the values for the active bounds into the right hand sides (rhs) of the equations defining the basic variables $x1, x3$, you will find that the resulting values for $x1, x3$ are *within* the respective bounds. Note that only the two higher dimensional inequalities led to the introduction of slack variables $s1, s2$.

Algorithmic Details: Finding the solved form of a bounded variable linear program can be rephrased as a search problem, where we have:

1. A given initial state, consisting of a system where the solved form invariants may be violated.
2. The specification of a solution state through the solved form invariants.
3. The operators:
   (a) $pivot(x_i, x_j)$
   (b) $toggle\_active\_bound(x_i)$ for non-basic variables

The non-determinism in the selection of the the operators and their arguments can be removed by the same rules that are employed in the original Simplex algorithm:

– In order to enter the basis, the type of the non-basic variable must be compatibe with the sign of the coefficient of the variable
in the objective function

- The leaving variable corresponds to the row in the working basis that imposes the tightest constraint on the entering variable
- The active bound of a variable may be toggled if the tightest constraint imposed on the variable through the working basis accomodates the change

A violation of the solved form is always detected by locating a basic variable that is out of its bounds. As the solved form will be computed incrementally, there will always be at most one such row and it will correspond to the $m+1$-st source constraint.

Theorem 1.

1. *The incremental solved form algorithm constitutes a decision algorithm for the satisfiability problem of a polyheral set*
2. *The incremental solved form algorithm detects implicitly fixed values*

Proof. The solved form obviously satisfies

$$\forall_{k \epsilon 1 \ldots m}\ l_k \leq inf(x_k) \leq \phi(x_k) \leq sup(x_k) \leq u_k \tag{3}$$

To establish the corresponding relation for the challenging row $i = m+1$, we interpret the linear combination of non-basic variables that defines the basic variable as artificial objective function.

$$x_{i_{[l,u]}} = \sum_{j | x_j \not\in basis} k_{ij} x_j + b_i \tag{4}$$

The evaluation $\phi(x_i)$ of x_i with respect to the BFS may give rise to a repair action consisting in the iterated application of the operators *pivot* and *toggle_active_bound* in order to decrement (increment) $\phi(x_i)$ until

1. $l_i < \phi(x_i) < u_i$: solved form established. Satisfiable.
2. None of the operators is applicable, optimality, thus

$$\phi(x_i) = inf(x_i) \ \ or \ \ \phi(x_i) = sup(x_i) \tag{5}$$

   (a) $\phi(x_i) = inf(x_i) > u_i$: unsatisfiable
   (b) $\phi(x_i) = inf(x_i) = u_i \rightarrow x_i = u_i$: fixed value
   (c) $\phi(x_i) = sup(x_i) < l_i$: unsatisfiable
   (d) $\phi(x_i) = sup(x_i) = l_i \rightarrow x_i = l_i$: fixed value

□

Practical Details: The incremental solved form for bounded variable linear programs forms the basis for the implementation of the CLP($\mathcal{Q}$) and CLP($\mathcal{R}$) systems distributed with the SICStus and Eclipse Prolog. The coverage is at least as complete as that of earlier CLP($\mathcal{R}$) implementations: The system incrementally 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 fixed values, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

It is coded in Prolog using *Attributed Variables* [8] which serve as direct access storage locations for properties associated with variables. At the same time, attributed variables make the unification part of a unification based language, Prolog in our particular case, user-definable within the language under extension [7, 9].

Empirics: In the following table we list the execution time ratio b/s between the solved form algorithm using bounded variables and a 'crippled' version which works like the original Simplex with one slack and a row for each inequality. The first two examples are from [4] computing the first and all solutions to the geometric covering problem where nine squares of unknown and different sizes are required fill an unspecified rectangle, the remaining ones are executions of a very simple minded branch and bound (BB) code on top of our solved form for some of the smaller examples from the MIPLIB mixed integer linear programming examples. Branch and bound is expected to benefit from generalized slacks because BB strengthens the original problem relaxation with simple inequalities like $x \leq \lfloor \phi(x) \rfloor$ and $x \geq \lceil \phi(x) \rceil$ when branching.

example	b/s
9 squares 1st	0.78
9 squares all	0.76
flugpl	0.48
stein15	0.68
sample2	0.70
bm23	0.73
egout	0.11
enigma	0.59
mod013	0.42
pipex	0.57
sentoy	0.77

On this collection, the solver with generalized slacks is roughly twice as fast, everything else held constant: same solver data structures, same base numeric (rationals), same machine. Basically the same ratios are obtained when computing with floating point numbers.

4 Determining Minimal Conflict Sets

In the section we will see how generalized slacks can be used to determine the reasons for the unsatisfiability of a set of constraints. We distinguish between equations, inequalities and disequalities, i.e. constraints of the form

$$\sum_{j=1}^{n} a_{ij} x_j \bowtie b_i \quad where \quad \bowtie \ \epsilon\{=, \neq, \leq, <, \geq, >\} \tag{6}$$

Interface: In the sequel we need the ability to refer to individual constraints by a symbolic name which we call a label. Figure 1 depicts the interface between the CLP($\mathcal{Q}$) solver kernel and the hierarchy manager part of IHCS(X,$\preceq$). The activation operation supplies the solver with the label of the constraint to be activated and the solver returns a conflict set (CS) where the label is an abstract data type not looked at by the solver, and the CS is a union of labels, possibly empty.

4.1 Failure Analysis for Equations

We solve systems of equations by Gaussian elimination. At any time, the set of variables is partitioned into basic and non-basic variables. The basic variables are expressed in terms of the non-basics. Upon the addition of a constraint it is *dereferenced* against the solved form. That is, references to the basic variables are replaced by their definitions. If the resulting expression is $0 = 0$, the constraint is entailed. The constraint is in conflict with the solved form if $0 = k, k \neq 0$. Otherwise, we solve for an arbitrary variable in the dereferenced expression and add this definition to the solved form.

We extend this scheme through the addition of a unique slack variable with bounds $[0, 0]$ to each equation. The basis for the validity of this operation is that one may substitute zero at any time for all such variables without changing the original problem statement. We call this special sort of slack variables *witness variables* after [6] where the very same trick was applied for a completely different purpose. The initial coefficients for the witness variables is immaterial, but 1 is convenient. As the solved form is manipulated, the coefficients change, and the witness variables track the dependencies between the constraints which originate from dereferencing and from pivot operations.

Example 4.

input constraint(s)	solved form
$a + b = 10$	$a = 10 - b - w_1$
$a + b = 10$	$a = 5 - \frac{1}{2}w_1 + \frac{1}{2}w_2$
$a = b$	$b = 5 - \frac{1}{2}w_1 - \frac{1}{2}w_2$

(7)

The two equations determine the values for a and b, as can be seen by substituting zero for w_i. Adding a third, incompatible constraint $a = 4$ dereferences into $-1 = -\frac{1}{2}w_1 + \frac{1}{2}w_2 + w_3$. That is $-1 = 0$ and the culprits are identified by the

witness variables $w_{1..3}$: removing any of the corresponding equations restores satisfiability.

Proposition2. *Witness variables with nonzero coefficients in a dereferenced equation identify the original constraints which are responsible for the entailment or unsatisfiability of the equation.*

4.2 Failure Analysis for Disequalities

By the use of a unique slack variables we turn each

$$\sum_{j=1}^{n} a_{ij} x_j \neq b_i \quad into \quad \sum_{j=1}^{n} a_{ij} x_j - b_i = s_{nz} \tag{8}$$

where s_{nz} may assume any value but zero. The resulting equation is dealt with as outlined in the previous section. In the implementation, an obvious optimization is to combine the slack s_{nz} with the witness variable for the equation.

4.3 Failure Analysis for Inequalities

After the solved form algorithm fails to inc/decrease the row that violates the solved form, as described in section (3.1), the following holds:

Proposition3. *The basic variables with non-zero coefficients in this row identify the constraints that are in conflict with the constraint the row represents itself.*

This is because the termination condition of the solved form algorithm, i.e. the non-applicability of the operators *pivot* and *toggle_active_bound* is, like in the original Simplex algorithm based on the signs, and in our case types, of the variables in the objective function. Upon termination, the value of the objective function is known *and* a corresponding set of non-basic variables is identified. A stronger result is:

Theorem 4. *Once the solved form algorithm detects unsatisfiability in (5), the constraints identified by the non-basic variables with nonzero coefficients constitute a* **minimal** *inconsistent subset of the set of constraints.*

We draw upon a result by deBacker [1], which extended Lassez's Quasi-Dual results [11] based on Fourier's theorem.

Theorem 5 Fourier 1827. *A set S of inequalities is inconsistent iff there exists a positive linear combination of inequalities which give $0 \leq k$, where $k < 0$.*

$$S : \left\{ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \right\}_{i \epsilon 1..m} \tag{9}$$

Theorem 6 Lassez 1990. *For S as defined above its quasi dual is:*

$$Q : \left\{ \begin{array}{l} \sum_{i=1}^{m} \lambda_i a_{ij} = 0 \\ \sum_{i=1}^{m} \lambda_i = 1 \\ \forall i \epsilon 1..m, \lambda_i \geq 0 \end{array} \right\}_{j \epsilon 1..n} \tag{10}$$

- *If Q is empty then S is solvable*
- *Otherwise let $M = min \sum \lambda_i b_i$*
  - *If $M \geq 0$, S is solvable*
  - *If $M < 0$, S is unsolvable*

Theorem 7 deBacker 1991. *When S is unsolvable, we have a* witness *vertex of Q corresponding to M. A subset of S given by the indices $\lambda_i \neq 0$ is a minimal inconsistent subset of S.*

Proof of theorem 4. We exhibit the correspondence between the dual problem that arises from the repair action in the solved form algorithm for an unsatisfiable constraint and the quasi dual formulation for the whole system of constraints. The *dual* problem [15] for an optimization problem in standard form

$$\begin{array}{l} \left\{ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \right\}_{i \epsilon 1..m} \\ maximize \sum_{j=1}^{n} c_j x_j \end{array} \tag{11}$$

is

$$\begin{array}{l} \left\{ \sum_{j=1}^{m} a_{ij} \lambda_j^{'} \geq c_i \right\}_{i \epsilon 1..n} \\ \forall j \epsilon 1..m, \lambda_j^{'} \geq 0 \\ minimize \sum_{j=1}^{m} b_j \lambda_j^{'} \end{array} \tag{12}$$

The quasi dual for the total system including the $m+1$-th row is

$$Q_{total} : \left\{ \begin{array}{l} \sum_{i=1}^{m+1} \lambda_i a_{ij} = 0 \\ \sum_{i=1}^{m+1} \lambda_i = 1 \\ \forall i \epsilon 1..m+1, \lambda_i \geq 0 \end{array} \right\}_{j \epsilon 1...n} \tag{13}$$

The dual and the quasi dual are related by:

$$\left\{ \underbrace{\sum_{i=1}^{m} \lambda_i^{'} a_{ij}}_{\geq c_j} + \lambda_{m+1} c_j = 0 \right\}_{j \epsilon 1...n} \tag{14}$$

Thus, except for $\sum_i \lambda_i = 1$, the dual and the quasi dual correspond. The presence of this sum in the quasi dual is just a technical trick to force a unique solution to the minimization problem in (10). Therefore, without it, and because in (12) the non-basic λ's are zero at the optimum, the λ's correspond under the scaling:

$$\lambda_{m+1}^{'} = 1 \; and \; \lambda_i^{'} = \lambda_i / \lambda_{m+1} \tag{15}$$

which yields of course the same incidence relation regarding minimal inconsistent subsets of S. □

Example 5. The first six of the following constraints are satisfiable. The addition of the seventh results in unsatisfiability.

$$\begin{array}{l|l} \text{source constraint} & label \\ \hline x+3y+2z \geq 5, & 1 \\ 2x+2y+z \geq 2, & 2 \\ 4x-2y+3z \geq -1, & 3 \\ x \geq 0, & 4 \\ y \geq 0, & 5 \\ z \geq 0, & 6 \\ 6x+5y+2z \leq 4 & 7 \end{array} \tag{16}$$

Put into standard form, the quasi dual reads:

$$\begin{array}{l} \begin{pmatrix} -1 & -2 & -4 & -1 & 0 & 0 & 6 \\ -3 & -2 & 2 & 0 & -1 & 0 & 5 \\ -2 & -1 & -3 & 0 & 0 & -1 & 2 \end{pmatrix} \bar{\lambda} = \bar{0} \\ \sum_{i=1}^{7} \lambda_i = 1, \forall i \epsilon 1..7, \lambda_i \geq 0 \\ minimize(-5\lambda_1 - 2\lambda_2 + \lambda_3 + 4\lambda_7) \\ gives : \bar{\lambda} = (\frac{1}{9}, 0, 0, \frac{5}{9}, \frac{2}{9}, 0, \frac{1}{9}) \end{array} \tag{17}$$

Taking $\frac{1}{9}sc_1 + \frac{5}{9}sc_4 + \frac{2}{9}sc_5 + \frac{1}{9}sc_7$ results in $0 \leq -1$, where sc_i is the i-th source constraint.
In our solved for we have the following: after the addition of sc_7, the solved form is violated at the corresponding slack variable $s_{7[\_,0]}$, where the upper bound is 0 but $\phi(s_7) = \frac{47}{13}$:

$$\begin{array}{ll} (*) & s_{7[\_,0]} = \frac{47}{13} + \frac{75}{13}x - \frac{19}{13}s_1 + \frac{4}{13}s_3 \\ & s_{2[\_,0]} = \frac{-15}{13} - \frac{22}{13}x + \frac{8}{13}s_1 - \frac{1}{13}s_3 \\ & y_{[0,\_]} = \frac{17}{13} + \frac{5}{13}x - \frac{3}{13}s_1 + \frac{2}{13}s_3 \\ & z_{[0,\_]} = \frac{7}{13} - \frac{14}{13}x - \frac{2}{13}s_1 - \frac{3}{13}s_3 \\ & s_{3[\_,\overline{0}]} \\ & x_{[\overline{0},\_]} \\ & s_{1[\_,\overline{0}]} \end{array} \tag{18}$$

The application of $pivot(y, s_3)$ reduces $\phi(s_7)$ to 1, but the solved form is still violated. None of the variables x, y, s_1 can enter the basis, thus $\phi(s_7) = 1 = inf(s_7)$.

$$\begin{array}{ll} (*) & s_{7[\_,0]} = 1 + 5x + 2y - s_1 \\ & s_{2[\_,0]} = \frac{-1}{2} - \frac{3}{2}x 1\frac{1}{2}y + \frac{1}{2}s_1 \\ & s_{3[\_,0]} = \frac{-17}{2} - \frac{5}{2}x - \frac{13}{2}y + \frac{1}{2}s_1 \\ & z_{[0,\_]} = \frac{5}{2} - \frac{1}{2}x - \frac{3}{2}y - \frac{1}{2}s_1 \\ & y_{[\overline{0},\_]} \\ & x_{[\overline{0},\_]} \\ & s_{1[\_,\overline{0}]} \end{array} \tag{19}$$

Reading off the coefficients from (*), we get: $\lambda^{'} = (-1, 0, 0, 5, 2, 0, 1)$ Note that the first component of this vector is negative because we assumed normalized

inequalities, i.e. every inequality expressed as $\sum k_{ij} x_i \leq b$, in the proof only. Application gives:

$$\begin{array}{rcl} -x-3y-2z \leq -5 & & -x-3y-2z \leq -5 \\ 5x \geq 0 & \stackrel{normalize}{\longrightarrow} & -5x \leq 0 \\ 2y \geq 0 & & -2y \leq 0 \\ 6x+5y+2z \leq 4 & & 6x+5y+2z \leq 4 \end{array} \rightarrow 0 \leq -1 \qquad (20)$$

5 Deactivation of Constraints

Again we organize this section after the classes of constraints we deal with. The basis for the correctness of the operations performed is the invertability of linear transformations. Space not permitting for more details, we only mention a) that entailed constraints have to be reactivated if no longer entailed because of the deactivation of an entailing constraint, and b) that deactivation has of course to deal with the transitive closure of consequences of fixed value detection/propagation.

5.1 Deactivating Equations

An equation is deactivated by solving for the corresponding witness variable. Between the introduction of the witness variable and the time we are about to relax the equation, the solved form was changed by linear transformations only, which is the guarantee that we can solve for the witness variable. Solving for a variable removes it from all the right hand sides of all basic variables where it occurs, and then we may abandon the row for the witness variable. Consider our example again:

Example 6.

input constraint(s)	solved form
$a+b=10$	$a = 10 - b - w_1$
$a+b=10$	$a = 5 - \frac{1}{2}w_1 + \frac{1}{2}w_2$
$a=b$	$b = 5 - \frac{1}{2}w_1 - \frac{1}{2}w_2$

(21)

To deactivate the second equation $a = b$, we solve for $w_2 = -10 + 2a + w_1$, substitute and drop the row for w_2:

Example 7.

input constraint(s)	solved form
$a+b=10$	$b = 10 - a - w_1$

(22)

Which is equivalent to the solved form for $a + b = 10$. If we deactivate the first equation instead, we have $w_1 = 10 - 2a + w_2$ and:

input constraint(s)	solved form
$a=b$	$b = a - w_2$

(23)

5.2 Deactivating Disequations

Recall that disequations are turned into equations via slacks. Deactivating a disequation is by solving for the witness variable for the corresponding equation, and dropping the row afterwards.

5.3 Deactivating Inequations

An inequation is deactivated by bringing the associated slack variable into the basis and by removing the corresponding bound. A variable is brought into the basis by pivoting it with the most constraining row in the basis. If there is no constraining row for a non-basic variable, we may simply drop the bound to be deactivated.

6 Conclusion

It turned out to be remarkable simple to meet the explanatory and defeasibility requirements of the IHCS scenario for the instantiation of the constraint solver component to CLP($\mathcal{Q}$). One concept, generalized slacks, provides both mechanisms. With regard to computational complexity, explanations are for free if we have to deal with inequalities free of fixed values only. If there are (implicitly) fixed values and/or additional equations, the extra cost for carrying along the witness variables is rewarded by the possibilities a) to deactivate the constraints later, and b) although not elaborated on here, to have backtracking without trailing in the constraint solver [6]. Our work shares objectives with [3]. Our improvement is in the addition of defeasibility to the thread and that we don't need an explicit inverse of the basis for CS computations.

With respect to applications we naturally envision classical dependency directed backtracking applications, but expect more rewarding results from the extra expressiveness and flexibility through the IHCS architecture.

Acknowledgments

This joint work was made possible by project SOL from the Human Capital and Mobility Programme of the European Union. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science, Research, and the Arts. The work at Faculdade de Ciências e Técnologia of the Universidade Nova de Lisboa was supported by Junta Nacional de Investigação Científica e Tecnológica (grant PBIC/C/TIT/1242/92).

References

1. Backer B.de, Beringer H.: Intelligent Backtracking for CLP Languages: An Application to CLP(R), in Saraswat V. & Ueda K.(eds.), *Symposion on Logic Programming*, MIT Press, Cambridge, MA, pp.405-419, 1991.

2. A. Borning, M. Maher, A. Martingale, and M. Wilson. Constraints hierarchies and logic programming. In Levi and Martelli, editors, *Logic Programming: Proceedings of the 6th International Conference*, pages 149–164, Lisbon, Portugal, June 1989. The MIT Press.
3. Burg J., Lang S.-D., Hughes C.E.: Finding Conflict Sets and Backtrack Points in CLP(R), in Hentenryck P.van(ed.), *Proceedings of the Eleventh International Conference on Logic Programming (ICLP94)*, MIT Press, Cambridge, MA, 1994.
4. Colmerauer A.: An Introduction to Prolog III, *Communications of the ACM*, 33(7), 69-90, 1990.
5. Dantzig G.B.: *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.
6. Hentenryck P.van, Ranachandran V.: *Backtracking without Trailing in CLP(R)*, Dept.of Computer Science, Brown University, CS-93-51, 1993.
7. Holzbaur C.: *Specification of Constraint Based Inference Mechanisms through Extended Unification*, Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, Dissertation, 1990.
8. 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.
9. Holzbaur C.: Extensible Unification as Basis for the Implementation of CLP Languages, in Baader F., et al., *Proceedings of the Sixth International Workshop on Unification*, Boston University, MA, TR-93-004, pp.56-60, 1993.
10. Imbert J.-L., Cohen J., Weeger M.-D.: An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-Interpreter for CLP, in *Special Issue: Constraint Logic Programming, Journal of Logic Programming*, 16(3&4), 235-253, 1993.
11. Lassez J.L.: Parametric Queries, Linear Constraints and Variable Elimination, *in Proceedings of the Conference on Design and Implementation of Symbolic Computation Systems*, Capri, pp.164-173, 1990.
12. Menezes F., Barahona P.: Preliminary Formalization of an Incremental Hierarchical Constraint Solver. In L. Damas L and M. Filgueiras (eds.), In *Proceedings of EPIA '93*, Springer-Verlag, Porto, October 1993.
13. Menezes F., Barahona P.: An Incremental Hierarchical Constraint Solver. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, MIT Press, 1995.
14. Menezes F., Barahona P.: Defeasible Constraint Solving. In *Proceedings of Iberamia 94*, McGraw-Hill Interamericana de Venezuela, Caracas, October 1994.
15. Murty K.G.: *Linear and Combinatorial Programming*, Wiley, New York, 1976.
16. Wilson M., Borning A.: Hierarchical Constraint Logic Programming. *J. Logic Programming*, 1993:16.