PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. 



# Applications of a Canonical Form for Generalized Linear Constraints

J.L. Lassez
IBM T.J. Watson Research Center

K. McAloon
CUNY Graduate Center

**Abstract**

Constraint methods are an important field of research in Artificial Intelligence and Operations Research. The integration of the constraint solving paradigm in logic programming languages has raised a number of new issues. Foremost is the need for a useful canonical form for the representation of constraints. In the context of an extended class of linear arithmetic constraints we develop a natural canonical representation and give procedures for computing it. Important issues encountered include negative constraints, the elimination of redundancy and parallelism. Using the canonical form, we then address the problem of incremental constraint propagation.

## 1 Introduction

In [Jaffar and Lassez] Jaffar and Lassez proposed a formal framework for reasoning with and about constraints in the rule based context of logic programming. Within this framework, one can generate a class of programming languages customized to deal with constraints over specific domains of computation. Examples include CLP(R) by Jaffar and Michaylov [Jaffar and Michaylov], which is designed for real arithmetic, Colmerauer's Prolog III which is designed for rational linear arithmetic, CIL by Mukai [Mukai ]oriented toward linguistic constraints and CAL by Aiba and Sakai [Aiba and Sakai ]which covers linear and nonlinear equality constraints over the complex numbers.

The technical results in the present paper will be relevant to such languages and the motivations, aims and methods may provide guidelines for other settings.

Constraint solving also plays a major role in Artificial Intelligence work. Recent papers by Davis [Davis] and Pearl [Pearl] provide further sources for current work in this area and on connections between AI and OR work. Considerable work has also been done on the design of programming languages to solve constraint satisfaction problems [Borning], [Sussman and Steele]. The work of Dincbas and van Hentenryck [van Hentenryck and Dincbas ], on AI problem solving in a Logic Programming framework with constraints brings several techniques together and has important applications in Operations Research. Further interesting work on parallel constraint solving is being carried out by Mizoguchi and his colleagues in the Intelligent Systems Lab at Tokyo Science University with applications to qualitative reasoning and expert systems. There is also a vast literature on mathematical programming. Moreover, this fundamentally important field is currently the subject of dramatic developments ranging from the work of Karmakar [Karmakar] and others on interior point methods in Linear Programming to the work on the complexity of the Theory of Real Closed Fields by Ben-Or, Kozen, and Reif [Ben-Or et al] and Kozen and Yap [Kozen and Yap]. To complete the picture, linear constraint solving has also been used as tool in programm verification work. Moreover, in this context inequations as well as inequalities arise naturally [Nelson] and the constraint solving algorithms developed in this work are directly relevant to languages such as CLP(R) and Prolog III.

It is clear that the design and usefulness of languages from the CLP class will depend heavily on this wealth of information from Mathematical Programming, Operations Research and Artificial Intelligence.

However, the criteria that an arithmetic constraint solver must satisfy in order to perform effectively in an Operations Research or AI context are not the same as those that are required for a solver embedded in a general purpose programming language. To use constraints as a primitive data structure to represent objects in a programming lannguage is not the same problem as designing data structures to represent constraints in an AI context. In short, designing languages to solve constraint satisfaction problems or optimization problems is a fundamentally different issue from using the constraint paradigm to design programming languages.

By way of example, a typical constraint solving algorithm will take as input a set of constraints and a function and produce as output the coordinates of a point for which the function reaches an extremal value relative to an order or partial order. But in our context we are interested in programs that reason about constraints as output as well as input: the elements of our domain of computation are implicitly defined as constraints. Problems of crucial importance for us such as the equivalence of sets of constraints do not seem to have attracted considerable attention in Operations Research or in Artificial Intelligence, although the relevant mathematics are well understood. In this respect, the pioneering work of Adler on equivalence of linear programs is the exception [Adler]. So our first requirement for a canonical form for sets of constraints is to standardize output and to answer the equivalence problem. However, going further, because of the requirements on a constraint solver for a CLP language, we want more than a syntactic facility; for use in a general purpose solver we would like the canonical form to play a role similar to that of the mgu in unification or canonical solved forms in equation solving and to have the requisite properties of incrementality and efficiency.

The language of extended linear constraints is comprised of *positive* constraints which are equations $ax = \beta$ and weak inequalities $ax \leq \beta$. Here $a$ denotes an $n$-dimensional vector of real numbers, $x$ denotes an $n$-dimensional vector of variables, $\beta$ denotes a real number and juxtaposition denotes inner product.

A basic *negative* constraint is a disjunction of inequations $a_i x \neq \beta_i, i = 1 \ldots n$. Using DeMorgan's Law and matrix notation, a negative constraint can be written $\overline{\{Ax = b\}}$ which denotes the set of points $x$ which lie in the complement of the set defined by the equations $Ax = b$. Conjunctions of negative constraints will be written $\overline{\{A_j x = b_j\}}$, $j = 1, \ldots, n$. Conjunctions of equality constraints will be written in matrix notation $Ax = b$ and similarly conjunctions of weak inequality constraints will be written $Ax \leq b$.

We also admit strict inequality constraints $ax < b$. In matrix form conjunctions of strict inequality constraints are written $Ax < b$. This is a form of hybrid constraint in that it can be reduced to the combined positive and negative constraints $Ax \leq b$, $\overline{\{a_i x = \beta_i\}}$, $i = 1, \ldots, n$.

Thus a set of *generalized linear constraints* consists of positive constraints $Ex = f$ and $Ax \leq b$, negative constraints $\overline{\{C_j x = d_j\}}$, and hybrid constraints $Gx < h$.

By way of example, the constraints $z \geq 0, x - y + z < 1, x \geq 0, y \leq 0, \overline{\{y = 0, z = 0\}}$ define a wedge shaped polytope with an edge and a facet removed.

We will define a notion of canonical form for sets of extended linear constraints and describe algorithms for computing the canonical form. These algorithms yield efficient polynomial time algorithms for testing the consistency of a set of generalized linear constraints and for computing the canonical form. The presence of negative constraints introduces new problems. The point sets defined by the constraints are no longer convex sets and so the methods of convex analysis and of linear programming do not apply directly. Negative constraints themselves are *disjunctions* of inequalities which enhances the expressive power of the constraint sets but which complicates the combinatorics of the situation. The key to dealing with these problems is the *independence of negative constraints*, a phenomenon which is central to Logic Programming, see [Lassez and McAloon ]. In this paper we give the procedures for computing the canonical form. In [Lassez and McAloon ]the mathematics necessary to verify the correctness of our algorithms can be found. We also discuss possible approaches to the implementation of the canonical form algorithm and its usefulness for dealing with problems such as incremental constraint propagation.

## 2 The Canonical Form Algorithm

We require some definitions. Suppose that $Ex = f, Ax, \overline{\{C_j x = d_j\}}$ is a feasible set of generalized linear constraints and that the positive constraints define a polyhedral set $P$; then a negative constraint $\overline{\{Cx = d\}}$ is said to be *relevant* if $\{x : Cx = d\} \cap P$ is not empty. In other words, a negative constraint is relevant if it is not already implied by the positive constraints.

Suppose that $P$ is a polyhedral set defined by a system of positive constraints; a negative constraint $\overline{\{Cx = d\}}$ is said to be *P-precise* if $\overline{\{Cx = d\}}$ is relevant and $\{x : Cx = d\} = Aff(P \cap \{x : Cx = d\})$. In other words, a relevant negative constraint $\overline{\{Cx = d\}}$ is precise relative to the polyhedral set $P$ if the affine space given as the complement of the negative constraint is equal to the affine closure of the intersection of this complement with $P$. By abuse of language if the positive constraints of a generalized system define the polyhedral set $P$, then negative constraints which are $P$-precise will be called precise with $P$ left understood.

A set of linear equations

$$\begin{aligned} y_1 &= a_{1,1}x_1 + \ldots + a_{n,1}x_n + c_1 \\ &\vdots \\ y_m &= a_{m,1}x_1 + \ldots + a_{m,n}x_n + c_m \end{aligned}$$

is said to be in *solved form* if the variables $y_1, \ldots, y_m$ and $x_1, \ldots, x_n$ are all distinct. The variables $y_1, \ldots, y_m$ are called the *eliminable variables* and the variables $x_1, \ldots, x_n$ are called the *parameters*. By abuse of language, equality constraints $Ex = f$ are said to be in solved form if the matrix $E$ is in row-echelon (or Gauss-Jordan) form. Further, we will say that a negative constraint $\overline{\{Cx = d\}}$ is in solved form if the equality constraint $\{Cx = d\}$ is given in solved form.

A set of generalized linear constraints is in *canonical* form if it is non-redundant and consists of (1) a set of equations in solved form with parameters $x$ defining the affine hull of the solution set (2) a set of inequality constraints $Ax \leq b$ which define a full dimensional polyhedral set $P$ in the parameter space and (3) a set of $P$-precise negative constraints in solved form.

A system of generalized linear constraints in canonical form is thus partitioned into three modules (E,I,N) where E is a set of equality constraints, I is a set of weak inequality constraints and N is a set of negative constraints. What we shall develop is an algorithm that maps generalized linear constraints to triples of this form in such a way that if two constraint sets define the same point set then they are mapped to the same triple (E,I,N).

We consider an example. Consider the constraints in four variables $x_1 + x_3 \leq x_4, x_1 + x_3 \leq 10, x_4 \leq x_3, x_2 \leq x_3 + x_4, x_3 \leq x_1 + x_4, 0 \leq x_2, \overline{\{x_4 = 0\}}$. The CanForm procedure will return

$$\begin{aligned} E &= \{x_1 = 0, x_3 = x_4\} \\ I &= \{x_4 \leq 10, x_2 - 2x_4 \leq 0, -x_2 \leq 0\} \\ N &= \{\overline{\{x_2 = 0, x_4 = 0\}}\} \end{aligned}$$

The constraints thus define a two dimensional point set, a triangle with a vertex removed.

In the definition of canonical from no mention is made of strict inequality constraints. As noted in the introduction, each strict inequality constraint $e_k x < \phi_k$ can be replaced by the pair $e_k x \leq \phi_k, \overline{\{e_k x = \phi_k\}}$. From the algorithmic point of view, we can suppose that this transformation has been made throughout; we return to this point later and show how to restore the strict inequality constraints at the end of the simplification process.

In what follows we will use Linear Programming and Gaussian Elimination procedures as part of decision algorithms. Hence, we require that arithmetical operations be carried out in sufficient precision. The necessary precision can be maintained within polynomial time resource, see e.g. [Schrijver]. Note that this problem does not arise in Prolog III or CHIP where infinite precision arithmetic is used.

To start, for the important special case where all the constraints are equality constraints, we have the Gauss-Jordan procedure, denoted GaJo($Ex = f$), for computing the equivalent solved form which is the canonical form in this case. The next case is to determine whether any of the weak inequality constraints in a system are in fact implicit equalities. Determining whether an equality $a_i x = \beta_i$ is implied by the constraints $Ax \leq b$ is a classic problem, e.g. [Luenberger] and can be solved by means of a linear program. Moreover, in this situation, the linear programs $min\{a_i x : Ax \leq b\}$ can be run in parallel. The procedure AffClo to compute the affine closure will call upon linear pro-

gramming subroutines and we assume that these routines will determine feasibility of the given constraints and return the optimum value of the objective function if it exists and $+\infty$ or $-\infty$ if the objective function is unbounded.

**Procedure** AffClo($Ax \leq b$)
Input: $Ax \leq b$, a set of $m$ positive constraints
Output: Equality constraints defining
$Aff(\{x : Ax \leq b\})$ in canonical form
For $i = 1, ..., m$
pardo
compute $min\{a_i x : Ax \leq b\}$;
return INFEASIBLE if applicable
if optimal value $\beta_i$ is returned set $flag_i = 1$
else set $flag_i = 0$
parend
Return GaJo($\{a_i x = \beta_i : flag_i = 1\}$).

As with the case of determining implicit equalities, determining if an inequality constraint is redundant is a classical problem in Linear Programming [Luenberger] and can again be solved by means of a linear program. Specifically, the constraint $a_1 x \leq \beta_1$ is redundant in the system $Ax \leq b$ if and only if the value returned by the linear program $max\{a_1 x : a_2 x \leq \beta_2, ..., a_n x \leq \beta_n\}$ is $\leq \beta_1$. Moreover, removing the redundant constraints in the full dimensional case will lead to a system which consists exactly of those constraints which define facets of the polyhedral set defined by the full collection of positive constraints. To facilitate our treatment of redundancy, we shall need some definitions. A positive constraint is said to be *trivial* if it is of the form $c_1 \leq c_2$ where $c_1, c_2$ are constants. We say that the constraint $a_i x \leq \beta_i$ is *syntactically redundant* in the system $Ax \leq b$ if it is trivial or if for some $j \neq i$ the constraint $a_j x \leq \beta_j$ is a positive scalar multiple of $a_i x \leq \beta_i$. A constraint is *semantically redundant* iff it is redundant but not syntactically redundant. We note that eliminating syntactic redundancy from a set of constraints can be done by sieving or sorting and so there are efficient sequential and parallel techniques for doing this. In [Lassez and McAloon] there is an analysis of semantically redundant constraints into a finer classification that is essential for verifying procedures which feature highly decomposable parallelism. In fact this classification is central to the development and the correctness of the canonical form algorithm.

These considerations lead to a natural definition. If the system of positive constraints $Ax \leq b$ defines a full dimensional set, we say the system is *incompressible* if for any system $A'x \leq b'$ which defines the same polyhedral set, every constraint of $Ax \leq b$ or a positive scalar multiple of it appears in the system $A'x \leq b'$. We have a procedure for computing an equivalent incompressible system of positive constraints in the full dimensional case.

**Procedure** Inc($Ax \leq b$)
Input: $Ax \leq b$, a feasible set of $m$ positive constraints which define a full dimensional polyhedral set
Output: An equivalent incompressible system
Call a parallel sieve or sort routine
to remove syntactic redundancy
For $i = 1, ..., m$
pardo
compute $max\{a_j x : a_j x \leq b_j, j \neq i\}$
if value returned is $\leq \beta_i$ set $flag_i = 1$
else set $flag_i = 0$
parend
Return $\{a_i x \leq \beta_i : flag_i = 0\}$).

Clearly, the procedure returns a system of non-redundant constraints. For correctness, it must also be shown that the parallel elimination of semantically redundant constraints yields a system equivalent to the original one. This is done in [Lassez and McAloon].

As for the issue of redundancy and negative constraints and the effect of the presence of negative constraints on redundancy among positive constraints and vice-versa, the situation can best be summarized in the following way: redundancy of positive constraints is independent of the presence of negative constraints and redundancy of precise negative constraints can be decided independently of the positive constraints by means of a parallel sieving procedure.

We now present a procedure for computing the canonical form in the general case.

**Procedure** CanForm($Ex = f, Ax \leq b, \overline{C_j x = d_j}$)
Input: $Ex = f, Ax \leq b, \overline{C_j x = d_j}, j = 1, \ldots, n$
Output: An equivalent system in canonical form
Call AffClo($Ex = f, Ax \leq b$) returning
$Affx = aff$ in canonical form;

eliminate variables throughout the
remaining constraints updating $Ax, C_j x = d_j$
For $j = 1, ..., n$
pardo
If $C_j x = d_j$ reduces to $0 = 0$ return INFEASIBLE
parend
Call Inc($Ax \leq b$) returning $A'x \leq b'$
For $j = 1, ..., n$
pardo
Call AffClo($A'x \leq b', C_j x = d_j$) returning $Aff_j x = aff_j$
parend
Sieve to eliminate redundancy among
the negative constraints
returning $Aff'_j x = aff'_j$
Return $Affx = aff, A'x \leq b', \{\overline{Aff'_j x = aff'_j}\}$

We have

**Theorem 1** *If two sets of constraints define the same point set the procedure CanForm returns the same equations to define the affine hull, the same inequality constraints (up to multiplication by positive scalars) and the same set of negative constraints.*

At this point, if strict inequality constraints are to be returned in the canonical form, let us note that a strict inequality corresponds to a pair $(ax \leq b, \overline{\{cx = d\}})$ where $ax \leq b$ is a positive weak inequality constraint in the canonical form and $\overline{\{cx = d\}}$ is a negative constraint such that the vector $c, d$ is a scalar multiple of the vector $a, b$. This pair can then be replaced by the strict inequality constraint $ax < b$. As is the use of linear programming and Gaussian elimination in the decision procedures of the CanForm algorithm, here too sufficient precision arithmetic is required. This variant on the canonical form algorithm is a natural one in the context of symbolic processing of generalized linear constraints and in the context of output constraints where strict inequality information can be significant.

If in the example above, the constraint $0 \leq x_2$ is sharpened to $0 < x_2$, then after transforming this constraint into the pair $0 \leq x_2, \overline{\{x_2 = 0\}}$, the CanForm procedure would return

$$E = \{x_1 = 0, x_3 = x_4\}$$
$$I = \{x_4 \leq 10, x_2 - 2x_4 \leq 0, -x_2 \leq 0\}$$
$$N = \{\overline{\{x_2 = 0\}}\}$$

The negative constraint $\overline{\{x_2 = 0, x_4 = 0\}}$ has been eliminated because it is now redundant. Since the vector $(1, 0, 0)$ is a scalar multiple of $(-1, 0, 0)$ the constraints $-x_2 \leq 0$ and $\overline{\{x_2 = 0\}}$ can be replaced by the strict inequality constraint $-x_2 < 0$.

# 3 Constraint Propagation and Constraint Programming

We now consider applications of the canonical form algorithm to constraint propagation. Linear arithmetic constraints arise constantly in constraint programming and AI situations, e.g. [Sussman and Steele], [Davis], [Pearl]. When determining the solvability and/or the solutions to a set of constraints the most usual strategy is to work forward in an incremental way by starting with the 'most constraining' conditions and propagating these constraints throughout the rest of the computation. Typically a constraint such as $x = 0$ is more constraining than a condition such as $x = y$ which is in turn more restrictive than $x \leq y$. In this example a more constraining condition corresponds to a set of smaller dimension in the solution space; the successive sets are of dimension 0,1 and 2. As a by-product the canonical form algorithm determines the dimension of the solution set - it is equal to the number of parameters of the system in canonical form - and returns a representation of the inequality and negative constraints in the smallest possible dimension. As for the negative constraints, the canonical form returns them in irredundant and precise form. By making the negative constraints precise relative to the set defined by the positive constraints the negative constraints are also reduced to their lowest dimension which enhances their information content. This technique also detects when a weak inequality is in fact a strong inequality and when a negative condition reduces to excluding a particular vertex or face of the polyhedral set in question.

The canonical form method yields a 'completeness theorem' in terms of the propagation of the equality, inequality and negative information contained in a system $S$ of generalized constraints.

**Theorem 2 (Constraint Propagation Theorem)** *Let $S$ be a system of generalized linear constraints with canonical form $(E, I, N)$. Let $y$ be the eliminable variables and $x$ the parameters of $(E, I, N)$. Then we have*

*(1)* $S \models ay = bx$ *iff* $E \models ay = bx$

*(2)* $S \models ax \leq \beta$ *iff* $I \models ax \leq \beta$

*(3)* $S \models \overline{\{Cx = d\}}$ *iff* $N \models \overline{\{Cx = d\}}$

*where* $\overline{\{Cx = d\}}$ *is a precise negative constraint.*

The proof uses the arguments that served to verify the correctness of the CanForm algorithm.

One application to constraint propagation of the canonical form algorithm that motivates us directly is its relevance to the constraint based logic programming language CLP(R). When non-linear constraints are introduced a delay mechanism is used to put consistency testing on wait until forthcoming instantiations of variables will make further parts of the constraint set linear at which point the constraint solver can be invoked. Thus it is important that maximum information from the linear constraints be propagated through the non-linear constraints. What is to be avoided is letting the system run on indefinitely or having it output a constraint system without providing a guarantee that it is solvable. By way of example, instead of $x \leq 3, x \geq 3, y \leq sin(x-3)+(x*x), y \geq (x*x*x)-18$ as output with no guarantee that this system is solvable, using the canonical form we would get $x = 3, y = 9$ as output.

Let us consider the computation cost of maintaining the canonical form. A number of possibilities should be considered to allow efficient implementations. That, is implementations where either the overhead is negligible, or if it is not, it is compensated by the fact that the added information obtained will help speed up the remainder of the computation. The most promising solution is to use parallelism in the way that has essentially been described in the previous section. For example, as all inputs tested in parallel for solvability and determination of the affine hull are very similar, we can assume that we obtain at the same time the information about solvability and about which inequalities are implied equalities. If there are none we have not incurred an overhead. If there are implied equalities, we pay the price of putting them in canonical form, but we may hope that this price will be offset by the fact that these new equalities will simplify and speed up the remainder of the computation.

It is easy to find examples where the use of the canonical form will lead to very substantial gains in efficiency, following the previous discussion. However it is also easy to find examples where it will be costly. This would be the case, for example, if all systems of constraints that we meet are already in canonical form! Here we would pay for the overhead of trying to generate new equalities for instance, when none exists. Of course at the end of the computation we will know that we have a canonical form which is useful information. But if for instance a lot of backtracking is involved the overhead can become prohibitive, and if the computation ends in failure, we have no added information to compensate.

Incrementality is another crucial property for constraint solvers used in programming languages and in applications which involve search and constraint propagation. The canonical form that we have developed here for generalized linear constraints has several advantages in this regard that we discuss now. For example, in contradistinction to unification and term matching where the most-general-unifier progressively becomes more and more complex, here the canonical form may become far simpler as constraints are added to the system. To illustrate this, let us consider the previous example where the constraints in canonical form are $x_1 = 0, x_3 = x_4, x_4 \leq 10, x_2 - 2x_4 \leq 0, -x_2 \leq 0, \overline{\{x_2 = 0, x_4 = 0\}}$; then if the constraint $x_4 + x_2 \geq 30$ is adjoined, the entire system now simplifies to $x_1 = 0, x_2 = 20, x_3 = 10, x_4 = 10$.

To consider the case where a positive inequality constraint $cx \leq \delta$ is introduced to the system, feasibility must be tested. With simplex methods and with interior point methods, feasibility will most naturally be treated as an auxiliary linear programming problem which will determine a vertex of the polyhedral set defined by the constraints in the feasible case. If this vertex satisfies the strict inequality $cx < \delta$ or if this vertex is non-degenerate, then testing for implicit equality is immediate and does not require an additional linear programming routine. Moreover, when a new inequality constraint $cx \leq \delta$ is adjoined to a system in canonical form, the following proposition shows that if the new constraint is *not* an implicit equality, then there are *no* implied inequalities in the new system. Thus if the new inequality constraint is consistent and not an implicit inequality, the augmented sys-

tem is full dimensional and the canonical form can be maintained by eliminating redundant constraints and maintaining the negative constraints in precise and irredundant form. More importantly when a new inequality constraint proves to be an implicit equality of the augmented system, then the theorem guarantees that further checks for implicit equality will prove fruitful. This means that the original polyhedral solution set has collapsed onto one of its faces potentially of arbitrarily small dimension.

**Theorem 3** *Suppose $Ax \leq b$ is a set of constraints that defines a full dimensional polyhedral set.*

*(1) If the constraint $cx \leq d$ is not an implicit equality of the combined set of constraints $Ax \leq b, cx \leq d$, then the combined constraints define a full dimensional polyhedral set.*

*(2) If the constraint $cx = d$ is an implicit equality of the combined system $Ax \leq b, cx \leq \delta$, then it is not the only implicit equality of the combined system.*

Let us note that a result similar to Part 1 of the above theorem and the strategy it suggests are used in Prolog III.

If constraints are maintained in canonical form, then absorbing an additional negative constraint $\overline{\{Cx = d\}}$ is straightforward: consistency of the augmented set reduces to a linear algebra problem, *viz.* checking that $Cx = d$ does not imply the current equality constraints. For after that consistency will be guaranteed by the fact that the negative constraint corresponds to removing a set of smaller dimension than that of the polyhedral set defined by the constraints in canonical form. So at this point the cost of maintaining the constraints in canonical form is that of putting the new negative constraint in precise form and checking for redundancy among the augmented set of negative constraints.

We have assumed the existence of a solver and used it as a black box. However there are many types of solver, and we may exploit their particular properties in order to compute more efficiently the canonical form. The optimal strategies for using the canonical form algorithm in CLP systems and other constraint oriented applications will have to be developed empirically and will depend crucially on the mathematical tests used for feasibility. For example, in Prolog III and in CLP(R), constraints are maintained in variants of the standard form of the Simplex Algorithm. At the data structure level this strategy is orthogonal to the canonical form approach developed in this paper which closely follows the geometry of the constraint solution set.

As a concluding note, in situations where a lot of backtracking is involved, the overhead of constantly maintaining a canonical form as opposed to a simple check for consistency could be prohibitive and at each failed path the added information provided by the canonical form is wasted. On the other hand, if constraint programming is added to a logic programming language without backtracking such as GHC [Ueda ]then the situation is completely different. *A propos*, in [Maher ], Maher has introduced a new class of committed choice languages with constraints and defined their logical and algebraic semantics. For such languages, the role played by a canonical form is an interesting topic for further research.

# Bibliography


[ Adler ]
I. Adler, The Core of a Linear Program, Technical Report, Department of Operations Research, Berkeley

[Aiba and Sakai ]
A. Aiba and K. Sakai, CAL: A Theoretical Background of Constraint Logic Programming and its Applications, to appear

[Ben-Or et al ]
M. Ben-Or, D. Kozen and J. Reif, The Complexity of Elementary Algebra and Geometry, *Proceedings of the 16th ACM Symposium on the Theory of Computing*, 1984

[Borning ]
A. Borning, The Programming Language Aspects of THINGLAB - A Constraint Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems* 3 (1981) 252-387

[ Colmerauer ]
A. Colmerauer, Equations and Inequations on Finite and Infinite Trees, *Proceedings of 1984 FGCS Conference*, Tokyo

[Davis ]
E. Davis, Constraint Propagation, *AI Journal*, 1988

[van Hentenryck and Dincbas ]
P. van Hentenryck and M. Dincbas, Forward Checking in Logic Programming, *Proceedings of the 1987 Logic*

*Programming Conference*, Melbourne, MIT Press
[ Jafar and Lassez ]
J. Jaffar and J-L. Lassez, Constraint Logic Programming, *Proceedings of POPL 1987*, Munich
[ Jaffar and Michaylov ]
J. Jaffar and S. Michaylov, Methodology and Implementation of a CLP System, *Proceedings of the 1987 Logic Programming Conference*, Melbourne, M.I.T. Press
[Karmakar ]
N. Karmakar, A New Polynomial Time Algorithm for Linear Programming, *Combinatorica* 4 (1984) 141-158
[ Karwan et al. ]
M.H. Karwan, V. Lofti, J. Telgen and S. Zionts, *Redundancy in Mathematical Programming*, Lecture Notes in Economics and Mathematical Systems 206, Springer-Verlag 1983
[Kozen and Yap ]
D. Kozen and C. Yap, Algebraic Cell Decomposition in NC, *Proceedings 19th ACM Symposium of the Theory of Computing*, 1987
[Lassez and McAloon ]
J.L. Lassez and K. McAloon, Independence of Negative Constraints, to appear
[Luenberger ]
D. Luenberger, *Linear and Non-Linear Programming*, Addison-Wesley, 1973
[Maher ]
M. Maher, Logic Semantics for a Class of Committed Choice Programs, *Proceedings of the 1987 Logic Programming Conference*, Melbourne, MIT Press
[Mukai ]
K. Mukai, Situations in Constraint, US-JAPAN AI Symposium, 1987, Tokyo
[Nelson] G. Nelson, Techniques for Program Verification, Xerox PARC Technical Report CSL-81-10, 1981
[Pearl ]
J. Pearl, Constraints and Heuristics, *AI Journal* 1988
[ Schrijver ]
A. Schrijver, *Theory of Linear and Integer Programming*, Wiley 1986
[Steele and Sussman ]
G. Steele and G. Sussman, CONSTRAINTS - a Constraint Based Programming Language, *AI Journal*, 1982
[Ueda ]
K. Ueda, *Guarded Horn Clauses*, MIT Press, to appear