In this metaphor, final survival corresponds to the existence of a solution to a problem expressed in the language of constraints. Better yet, determining a solution amounts to finding a 'creature' capable of withstanding such ruthless constraints.

Constraint Logic Programming Languages

Jacques Cohen

Constraint Logic Programming (CLP) is an extension of Logic Programming aimed at replacing the pattern matching mechanism of unification, as used in Prolog, by a more general operation called constraint satisfaction. This article provides a panoramic view of the recent work done in designing and implementing CLP languages. It also presents a summary of their theoretical foundations, discusses implementation issues, compares the major CLP languages, and suggests directions for further work.

ILLUSTRATION: DIANA ZELVIN
Imaginary Scenario
Handcuffed and gagged, Houdini crouches inside a tight box pierced by threatening swords. Horrified, he stares at the approaching tip of yet another sword being thrust into the box.

To The Reader
Don't fret over the fate of our besieged character. In the world of non-determinism and constraints, he will regain life once the killer sword is withdrawn. Nonetheless, his destiny may remain uncertain until the supply of swords is exhausted.
Logic Programming (LP) can be viewed from two perspectives: one related to mathematical logic and automated theorem proving, the other to the development and usage of programming languages based on logic [11]. Similarly, the motivation for developing Constraint Logic Programming (CLP) can be viewed from these two perspectives.

Within the mathematical logic context, CLP represents an effort to establish a class of first-order theories which preserve the basic computational properties of Horn-clause logic. From the programming languages point of view, the purpose is to establish a class of logic programming languages in which the variables can have values in a diverse set of domains including trees, booleans, reals, rationals, lists, etc. [16, 30].

The objectives of this article are:

- to provide a survey of the recent work done in designing and implementing CLP languages,
- to present their theoretical foundations, and
- to describe the current research being done in this exciting new area of computer science.

This article is directed toward users, implementors and designers of LP languages; a basic knowledge of Prolog is assumed. Two of the main CLP representatives will be emphasized and described: Prolog III and CLP (R). The first has been proposed by Colmerauer, the originator of Prolog, and the head of the Groupe d'Intelligence Artificielle in Marseille, France; the second was developed by a logic programming team at IBM Yorktown Heights. Both languages have been implemented. Two other groups, one European [19, 25], the other Japanese [1], have proposed and are currently implementing similar languages.

The work described in this article is quite recent, it began around 1986. But the use of constraints in Artificial Intelligence had been proposed earlier by Steele and Sussman in 1980 [43] Constraints have also been utilized in a graphic language developed by Borning in 1981 [5]. The essence of their work is illustrated by the following example. Consider the formula for translating temperatures from Celsius to Fahrenheit (and vice versa):

\[ F = 1.8 * C + 32 \]

Steele and Sussman developed programming techniques for computing \( F \) given \( C \), or \( C \) given \( F \), using a variant of lazy evaluation known as local propagation. Operations are performed only when both operands are known numerical values; this is done by traversing linked lists representing the formula. When both \( F \) and \( C \) are unknown the result becomes the formula itself.

It is the author's opinion that the embryonic attempts to introduce constraints in Prolog are due to Colmerauer with his design and implementation of Prolog II [14]. In that language the occur test becomes unnecessary since the domain of that language is that of infinite trees [13, 15]. The unification of such trees is performed by solving systems of equations involving tree-valued variables. Backtracking is initiated whenever the equations have no solution; the equations are called constraints. "Disequations" were also introduced by Colmerauer, using the predicate \( \text{dif}(T_1, T_2) \), where \( T_1 \) and/or \( T_2 \) can be unbound variables. If at some future point \( T_1 \) and \( T_2 \) become equal, then this predicate will cause backtracking to occur. In the language of constraints, \( \text{dif} \) adds the disequation \( T_1 \neq T_2 \) to the current set of constraints.

For the sake of establishing a historical record, it should be mentioned that the language Absys I (developed in the late 1960s at the University of Edinburgh) already incorporated some of the features currently used in CLP languages [20, 21]. Although non-determinism, the elimination of the occur test, and testing for satisfiability of systems of equations were already present in Absys I, the SL theory of Kowalski and Kuehner [32], and the illuminating set of examples of Prolog usage given by Colmerauer and his group were to appear only a few years later. The author considers the latter developments essential in providing the impetus for the establishment of LP as practiced today.

Having presented this brief historical outline, the rationale for introducing constraints in Prolog is considered. A natural extension of viewing unification as a method of solving systems of equations involving trees is the generalization to new domains in which equations can be tested for solvability. The following example motivates the introduction of constraints. In Prolog, the equality:

\[ 1 + x = 3 \]

results in a failure, since the operation \( + \) is considered as an unevaluated function symbol and the unification algorithm fails!

In the past, there have been two unsatisfactory ways to circumvent this problem. The first is to use Peano's axioms, (i.e., to define predicates using the successor function). The predicate:

\[ \text{plus}(x, y, z) \]
means that \( Z \) is the sum of \( X \) plus \( Y \), and the axioms for addition are:

\[
\begin{align*}
\text{plus}(0, Y, Y) & , \\
\text{plus}(s(X), Y, s(Z)) & :- \text{plus}(X, Y, Z).
\end{align*}
\]

The equation \( 1 + X = 3 \) is expressed by the query:

\[- \text{plus}(s(0), X, s(s(0))).\]

resulting in \( X = s(0) \).

The use of the successor function is obviously only feasible for small integers. Programming with Peano's axioms (like coding Turing Machines) is not only anachronistic but incompatible with the very high-level-language characteristics of Prolog.

The second (unsatisfactory) way to bypass the difficulties of expressing the equality of arithmetic expressions is to write special predicates using tests and assignments which attempt to determine the values of the variables. In the example, this results in:

\[
\text{X is 2}
\]

where the predicate \( \text{is} \) represents an assignment, always requiring a variable as its left operand.

A (still-unsatisfactory) variant of the above is to utilize the predicate \( \text{freeze}(X, P) \) which postpones the interpretation of \( P \) until \( X \) is bound. One could then write:

\[
\text{plus}(X, Y, Z) :- \text{freeze}(X, \text{freeze}(Y, Z \text{ is } X+Y)), \\
\text{freeze}(Z, X \text{ is } Z+Y)), \\
\text{freeze}(X, Y \text{ is } Z+X)).
\]

A general and clean solution to this problem is to replace unification of trees by constraints such as the equality of arithmetic expressions. For example, the constraint:

\[
2 + X = Z + 3
\]

will be handled as in algebra. The actual values for \( X \) or \( Z \) may be computed later when more constraints are added. “Dead end” situations occur when the system of constraints is unsatisfiable, and such situations result in backtracking, just as in Prolog.

Program variables in CLP behave like mathematical variables, and the proposed languages cover a variety of domains (e.g., reals, booleans), relations (e.g., equalities, inequalities), and operations (e.g., addition, multiplication) that are used in expressing systems of constraints. A unique characteristic of the CLP class of languages is that their semantics share a large common nucleus. (This nucleus will be presented in the sections entitled “Basic Theoretical Results” and “Overview of Theoretical Foundations of CLP Languages.”) Most other language classes do not enjoy this property. For example, apparently similar imperative languages (e.g., Pascal and Ada) or functional languages (e.g., Lisp and Scheme) can differ markedly in their denotational and operational semantics.

In the next section we present a few examples of CLP programs to illustrate the need for constraint languages (A fairly large body of examples appears in [17, 24] and the interested reader is encouraged to examine them).

### A Few Examples

The first example presented is the classic program for computing Fibonacci series. Before presenting it, it is helpful to consider its Prolog counterpart:

\[
\begin{align*}
\text{fib}(0, 1). \\
\text{fib}(1, 1). \\
\text{fib}(N, R) :- N \text{ is } N-1, \\
\text{fib}(N-1, R1), \\
\text{fib}(N-2, R2), \\
\text{R is R1+R2}.
\end{align*}
\]

The \( \text{is} \) predicate prevents the program from being invertible: the query \( \text{fib}(10, X) \) succeeds in producing \( X=89 \) as a result, but the query \( \text{fib}(Y, 89) \) fails, since \( N \) is unbound and the assignment to \( N1 \) is not performed. Note that if we had placed the predicate \( N \geq 2 \) prior to the first recursive call, the query \( \text{fib}(Y, 89) \) would also fail, since the value of \( N \) is unbound and the test of inequality cannot be accomplished by the Prolog interpreter.

The modified CLP version of the program illustrates the invertibility capabilities of CLP interpreters:

\[
\begin{align*}
\text{fib}(0, 1). \\
\text{fib}(1, 1). \\
\text{fib}(N, R1+R2) :- \ N \geq 2, \\
\text{fib}(N-1, R1), \\
\text{fib}(N-2, R2).
\end{align*}
\]

where \( N \geq 2 \) is a constraint.

The query:

\[- \text{fib}(10, \text{fib})\]

yields \( F1b = 89 \), and the query:

\[- \text{fib}(N, 89)\]

yields \( N = 10 \).
The second example presented is a sorting program. For the purposes of this article, it is unnecessary to provide the code of this procedure (see [43]); qsort ([1, 2, 3], 3) sorts an input list L by constructing the sorted list S.

When L is a list of variables, a Prolog interpreter would fail since unbound variables cannot be compared using the relational operator \( \leq \). In CLP, the query:

\[ \textit{?- qsort([X1, X2, X3], S).} \]

yields as result \( S = [X1, X2, X3] \), \( X1 \leq X2, X2 \leq X3 \). When requested to provide all solutions, the interpreter will generate all the permutations of \( L \) as well as the applicable constraints.

The third and final example is the determination of the equations representing circles passing through two points. The following unit clause specifies a point of coordinates \( X, Y \) lying on the perimeter of a circle whose center has coordinates \( A \) and \( B \).

\[ \textit{on-circle (p(X,Y),c(A,B,(X-A)^2 + (Y-B)^2)).} \]

The query:

\[ \textit{?- on-circle (p(7,1),C), on-circle(p(0,2),C).} \]

yields the two constraints:

\[
-2B + 14A - 46 = 0 \\
-R^2 + 50A^2 - 350A + 625 = 0
\]

The first equation represents the straight line containing the loci of the centers of all circles passing through the two given points. The second equation enables the calculation of the corresponding radii.

This example involves non-linear equations and cannot be processed using either of the two major languages. The above results were obtained using a CLP meta-level interpreter developed at Brandeis University.

This section concludes by noting that constraints can be used to increase the purity of programs by avoiding \textit{cuts} and the \textit{not} operator. For example, the program:

\[
p :- q', alpha. \\
p :- beta.
\]

with \( q' \) being the complement of \( q \) (e.g., if the main predicate of \( q \) is \( = \) (or \( \leq \)), that of \( q' \) becomes \( \neq \) (or \( \geq \)). The same artifice can be utilized to avoid certain uses of the \textit{not} operator.

\section*{Meta-Level Interpretation and Abstract Machine}

Two approaches for defining the inference mechanism utilized in constraint logic programming languages are presented in this section. The first is a meta-level interpretation allowing the description of interpreters for the languages (such as Lisp or Prolog) using the languages themselves. The second approach is based on automata theory.

In Prolog, the meta-level interpreter for pure programs consists of a few lines of code. The procedure \textit{solve} has as a parameter a list of Prolog goals to be processed. The interpreter assumes that the program rules are stored as unit clauses:

\[
\textit{clause(Head,Body)}. \\
\]

each corresponding to a rule:

\[
\textit{Head :- Body}. \\
\]

where \( \text{Head} \) is a literal and \( \text{Body} \) is a list of literals. Unit clauses are stored as:

\[
\textit{clause(Head,[])}. \\
\]

The interpreter is:

\[
\textit{solve([]).} \\
\textit{solve([Goal|Restgoal]):- solve(Goal), solve(Restgoal).} \\
\textit{solve(Goal):- clause(Goal,Body), solve(Body).} \\
\]

In CLP languages, a rule is represented by:

\[
\textit{clause(Head,Body,Constraints)} \\
\]

corresponding to a rule:

\[
\textit{Head :- Body} \{\text{Constraints}\}. \\
\]

The modified procedure \textit{solve} contains three parameters:

1) the list of goals to be processed,
2) the current set of constraints, and
3) the new set of constraints obtained by updating the previous set.

The meta-level interpreter for CLP, written in Prolog, is shown in Figure 1.

The heart of the interpreter is the procedure merge-constraints, which merges two sets of constraints: the previous constraints, Previous_C, and the constraints introduced by the current clause, Current_C. If there is no solution to this new set of constraints, the procedure fails; otherwise, it simplifies the resulting constraints, and it binds any variables which have been constrained to take a unique value. For example, the constraints \( X \leq 0 \land X \geq 0 \) simplify to the constraint \( X = 0 \), which implies that \( X \) can now be bound to 0.

The design considerations which influence the implementation of this procedure will be discussed in the section entitled “Design and Implementation Issues.”

Note that the controversial unit LIPS (logical inference steps per second), often used to estimate the speed of Prolog processors, loses its significance in the case of a constraint language. The number of LIPS is established by counting how many times per second the procedure clause is activated; in the case of CLP, the time to process clause and merge-constraints may vary significantly depending on the constraints being processed.

The second approach to describing the inference mechanism for a CLP language is to consider an abstract machine of the type suggested by Colmerauer in Prolog III. It resembles a push-down automaton [27] whose stack is updated whenever a program rule is applied.

We define a constraint-machine state \( \sigma \) as the triplet:

\[
\sigma = (W', t_0, ..., t_n, S)
\]

in which:

- \( W' \) is the list of variables which appear in the original query,
- \( t_0, ..., t_n \) is a list of terms (goals), and
- \( S \) is a list of current (satisfiable) constraints.

An inference step for the machine consists of making a transition from the state \( \sigma \) to a state \( \sigma' \) by applying a program rule:

\[
s_0 \rightarrow s_1, ..., s_m, R \quad (m > 0)
\]

in which the \( s_i \)'s are terms and \( R \) is the set of constraints specified by the rule. The new state \( \sigma' \) becomes:

\[
\sigma' = (W', s_1, ..., s_m, t_0, S' \cup R \cup (s_0 = t_0)),
\]

if \((S' \cup R \cup (s_0 = t_0))\) is satisfiable. Otherwise, another rule has to be tried nondeterministically. Note that the query variables \( W' \) remain unchanged throughout the execution of a program.

As in the case of Prolog, there are two types of nondeterminism that arise in the sequential interpretation of CLP programs: the first is the choice of an applicable clause in the program, and the second is the selection of the literal in the goal list that will be processed first.

It should be noted that the expression:

\[
S' \cup R \cup (s_0 = t_0)
\]

corresponds to the result of the procedure merge-constraints described earlier in this section. It will be shown later that the list \( W' \) is useful in the presentation of the output of a solution. A solution is found when a final state is reached; it has the form:

\[
\sigma_f = (W', \epsilon, Final_Constraint)
\]

where \( \epsilon \) is the empty sequence. In this case, the Final_Constraint is simplified, and then output in terms of the variables in \( W' \) and, if necessary, some additional ones.
Basic Theoretical Results

The objective of this section is to provide the reader with a summary of the fundamental results applicable to logic programs (see [2, 351]). It will be shown in the next section that these results remain applicable to a wide class of constraint logic programs.

Consider a logic program P consisting of (definite) Horn clauses such as:

$$p \leftarrow q_1, ..., q_n \quad (n \geq 0)$$

in which p and the qi's are all positive literals. Recall that the logic reading of the above is:

$$p \text{ is true if } q_1, q_2, ..., q_n \text{ are true}$$

or, in the case where n = 0, simply:

$$p \text{ is true}.$$

A query Q is a conjunction of positive literals t_1 ... t_m with m \geq 1. Let us initially assume that all literals t_i in Q are ground, (i.e., do not contain variables).

The first of the main theoretical results is due to Hill [26] and it can be summarized as follows:

(a) Equivalence of the logical and operational meaning of programs: a correct interpreter of the program (i.e., an SLD breadth-first theorem prover) will provide a yes answer to Q if and only if Q is a logical consequence of P.

The second main result reinforces the first by providing yet another meaning for logic programs, based on set theory. Consider the set S_0 of the unit clauses in P. If these clauses contain variables, then each such clause represents a potentially infinite number of ground literals where the variables are replaced by every possible ground term (and their combinations) appearing in P, (i.e., the so-called elements of the Herbrand Universe). Let S_0 be the set of ground instances of unit clauses. Consider now the ground instances of the rules in P in which all the qi's in their bodies are elements of S_0. Let T(S_0) be the set of heads of these ground rules. Basically this operation amounts to finding all ground terms corresponding to the application of one rule. Note that S_0 \subseteq T(S_0) since unit clauses have empty bodies.

Let us define T \uparrow n as the result of applying T to S_0 n times. Note that:

$$T \uparrow i \subseteq T \uparrow (i + 1)$$

and let us define:

$$T \uparrow \omega = \bigcup_{i=1}^{\infty} T \uparrow i$$

This is the set of all ground terms that can be proved from the unit clauses by applying some finite number of rules. It can be shown that T \uparrow \omega is the least-fixed-point of T, i.e., the smallest set S such that:

$$T(S) = S$$

The second important theoretical result due to Van Emden and Kowalski [46] can be stated as follows:

(b) Given a program P and a ground query Q, then (a) applies (i.e., Q is a logical consequence of P) if and only if the conjunctions of Q are members of the least-fixed-point set T \uparrow \omega of P, and conversely.

It can be shown that the results (a) and (b) generalize to include queries containing variables.

Note that these results are only applicable to (definite) Horn-clause programs P and queries Q yielding a yes answer. The results that explain the meaning of the program P and the query Q when the interpreter provides a no answer are now summarized. Clark [10] defined the completion of a Prolog predicate to be a non-Horn formula, that gives both necessary and sufficient conditions for the predicate to be true. For example, the completion of:

$$p \leftarrow$$

is the formula:

$$p \equiv (q_1 \land ... \land q_n) \lor ... \lor (r_1 \land ... \land r_m)$$

combined with the axioms specifying the equality of terms.

The following property corresponding to (a) is due to [10] and to [28]:

(a') A correct interpreter of the program (i.e., an SLD breadth-first theorem prover) will provide a yes answer to Q if and only if \neg Q is a logical consequence of the completion of P.

To establish the property corresponding to (b) for the case of no answers, we again use the operator T. First consider the set H_0, called the Herbrand base, whose elements are the literals of P in which variables are replaced by elements of the Herbrand Universe. Let T \downarrow n be defined as the result of applying T to H_0 n times. Note that:

$$T \downarrow i \supseteq T \downarrow (i + 1)$$
and let us define:

$$T \downarrow \omega = \bigcap_{i=1}^{\infty} T \downarrow i$$

It should be noted that $T \downarrow \omega$ is not necessarily a fixed point of $T$.

The property (b') corresponding to (b) for programs yielding a no answer is (see [28]):

(b') $\neg Q$ is a logical consequence of the completion of $P$ if and only if some conjunct of $Q$ is not a member of the set $T \downarrow \omega$.

Besides programs yielding yes or no answers, there are those which loop. The halting problem tells us that we cannot hope to detect all the programs which will eventually loop. Let us consider, as an example, the program $P_1$:

$$p(a),
p(b): -p(b).$$

As expected, the queries:

$$Q_1 : p(a), Q_2 : p(c)$$

yield respectively:

yes and no

since $p(a)$ is a consequence of $P_1$, and $\neg p(c)$ is a consequence of the completion of $p_1$, i.e.:

$$p(X) = (X = a) \lor (X = b \land p(b))$$

But the interpreter will loop for the query $Q_3 : p(b)$, or when all solutions of $Q_4 : p(X)$ are requested.

The sets $T \uparrow \omega$ and $T \downarrow \omega$ for $P_1$ are depicted graphically in Figure 2a). The corresponding graphical set representation for any logic program $P$ is depicted in Figure 2b).

The complement of the set $T \downarrow \omega$ defines the so-called finite failure set $F$. If a query $Q$ is in $F$ then the execution of program $P$ will yield a no answer.

A final example illustrates the semantic problems involved in using the not operator, as defined in Prolog by:

```prolog
not(X) :- X,!,fail.
not(X).
```

This is a convenient way of implementing negation which is not yet thoroughly understood: no rigorous semantic theory exists when not is employed in the body of clauses. From a logical point of view such clauses are not Horn clauses and properties (a) and (b) are no longer applicable.

Consider the program $P_2$:

$$p(a).
q(b).$$

The query $\text{not}(p(X)), q(X)$ yields a wrong no answer, since $X$ is initially bound to $a$, $\text{not}(p(X))$ fails, and the binding of $X$ is lost! By reversing the order of the elements of the query to $q(X), \text{not}(p(X))$, $X$ is correctly bound to $b$ and $\text{not}(p(X))$ succeeds.\(^1\)

**Overview of Theoretical Foundations of CLP Languages**

The theoretical underpinnings that have been proposed to explain the meaning of constraint logic programs are summarized in this section. Two approaches have been explored. Jaffar and Lassez [29] provide a meta-theory insuring that propositions $(a, a')$ and $(b, b')$ remain applicable in the case of constraint programs, provided the domains being considered satisfy certain conditions that

---

\(^1\)It has been shown that properties $(a, a')$ and $(b, b')$ are still applicable for guard queries containing positive and negative literals [33].
will be explained below. The other approach, taken by Colmerauer [17], is to describe the meaning of programs by establishing the relationship between rewriting rules representing the programs and the set of solutions obtained by applying these rules.

**Jaffar and Lassez's Approach**

This approach replaces the unification of trees (terms) by considering its two components. The first is a theory $\tau$, which, in the case of trees, corresponds to a set of axioms defining the equality of terms. Note that these axioms are expressed by general predicate-calculus formulas (see [10]). The second component is a model $R$, i.e., a mathematical description of the objects and relations that are axiomatized by the theory $\tau$. In the case of Prolog, the model is the set of finite trees with the usual equality relation and the theory is Clark’s equality theory.

Both $\tau$ and $R$ are defined in terms of two (finite) sets:

- $\Pi$: the set of predicate symbols, e.g., in the case of trees $\{=\}$
- $\Sigma$: the set of function symbols, e.g., in the case of trees all (unevaluated) function symbols

Another example of an interesting constraint domain is that of infinite trees and $\text{dif}$, as proposed by Colmerauer for Prolog II [14]. In that case:

- $R$: infinite trees
- $\tau$: axioms defining the equality and disequality of infinite trees
- $\Pi$: $\{=, \neq\}$
- $\Sigma$: as in the previous example

A basic property linking $\tau$ and $R$ is: a constraint is satisfiable in the theory $\tau$ if and only if it is satisfiable in its model $R$.

Jaffar and Lassez have shown that properties (a, $a'$) and (b, $b'$) of the previous section remain valid for a new constraint domain $D$ provided that:

1. The theory $\tau_D$ is satisfaction-complete, i.e., every constraint is either provably satisfiable or provably unsatisfiable.
2. The model $R_D$ is solution-compact, i.e., each element of the domain $D$ can be specified by a (potentially infinite) number of constraints, and the complement of any constraint can be specified by a (possibly infinite) number of constraints.

Table I contains four examples of constraint domains satisfying (or not) satisfaction-completeness and solution-compactness.

<table>
<thead>
<tr>
<th>Example of non-satisfaction-complete:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain $D$: integers</td>
</tr>
<tr>
<td>Theory $\tau$: predicate ${=}$</td>
</tr>
<tr>
<td>functions ${+,*}$</td>
</tr>
<tr>
<td>$x^* + y^* = z^*$ is not known to be satisfiable.</td>
</tr>
<tr>
<td>More generally, there is no algorithm for solving diophantine equations [38], so there is no (recursive) axiomatization $Z$ of the integers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example of satisfaction-complete:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain $D$: reals</td>
</tr>
<tr>
<td>Theory $\tau$: predicates ${=, \leq, \geq, &lt;, &gt;, \neq}$ functions ${+, *}$</td>
</tr>
<tr>
<td>Tarski has shown that there is an algorithm for deciding the satisfiability of polynomial equations and inequations over the reals numbers [45].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example of solution-compact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain $D$: reals</td>
</tr>
<tr>
<td>Theory $\tau$: predicates ${=, \leq, \geq, &lt;, &gt;}$ functions ${+, *}$</td>
</tr>
<tr>
<td>$\pi$ can be defined by a potentially infinite number of predicates.</td>
</tr>
<tr>
<td>$3 &lt; \pi &lt; 4$</td>
</tr>
<tr>
<td>$3.1 &lt; \pi &lt; 3.2$</td>
</tr>
<tr>
<td>$3.14 &lt; \pi &lt; 3.15$</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example of non-solution-compact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain $D$: reals</td>
</tr>
<tr>
<td>Theory $\tau$: predicate ${=}$</td>
</tr>
<tr>
<td>function $[+]$</td>
</tr>
<tr>
<td>Unable to define $\pi$ with a finite (or infinite) number of constraints.</td>
</tr>
</tbody>
</table>
calculus axioms specifying the equality (or disequality) of finite (or infinite) trees.

The beauty of the Jaffar and Lassez meta-theory is that they have established conditions under which the basic theorems of logic programming remain valid, provided that the set of proposed axioms specifying constraints satisfy the described properties.

A convenient (although incomplete) taxonomy for CLP languages is to classify them according to their domains or combinations thereof. One can have CLP (Rationals), or CLP (Booleans, Reals). Prolog can be described as CLP (Trees) and CLP (R) as CLP (Reals, Trees). A colleague of the author's (T. Hickey) has pointed out the existence of CLP(1); it corresponds to parameterless Horn clauses or to a puzzling language in which only variables are allowed without specifying their domain. A complete specification of CLP language would also have to include the predicates and operations allowed in establishing valid constraints (see examples in Table I).

From the language-design perspective, the designer would have to demonstrate the correctness of an efficient algorithm implementing the test for constraint satisfiability. This is equivalent to proving the satisfiability of the constraints specified by the axioms.

Colmerauer's Approach

This approach is based partly on the analogy between top-down parsers and SLD theorem provers. The reader is referred to [11], where this analogy is described in greater detail. Essentially, a literal is considered as a non-terminal and a logic program is viewed as a sequence of context-free rewriting rules [27]. Unit clauses correspond to rewriting the corresponding non-terminal into the empty sequence $\varepsilon$. A query corresponds to a sequence of non-terminals, and it succeeds if those non-terminals can be rewritten into $\varepsilon$ (i.e., erased) using the program rules. In other words, every non-terminal has a finite parse tree whose leaves are $\varepsilon$. These trees are the counterpart of the proof trees in theorems proving and the abstract machine (in the section entitled “Meta-Level Interpretation and Abstract Machine”) is the parsing push-down automaton. It should be noted that a rule containing variables actually represents a potentially infinite number of rules in which the variable is replaced by all possible values that can be assigned to it. (This situation is not unlike the one evoked in the previous section in describing the $T \uparrow$ operator).

Colmerauer proposes a self-contained theoretical description of the semantics of a constraint language assuming that notions such as: domain of the rationals, addition, multiplication, etc., are well known and need no axiomatization.

In his approach, a term is constructed using known operations (or partial operations) on variables, constants and function symbols in a given domain. In Prolog III the basic domain is that of infinite trees. A program rule consists syntactically of a pair: rewriting rule {constraints} in which: (1) a rewriting rule specifies that a term can be rewritten into a (possibly empty) sequence of terms, and (2) a constraint is a relation on terms.

Operations are partial since they are chosen by the language designer to be applicable only to selected subsets of the domain being considered. For example, the operation $x + y$ is only defined when $x$ and $y$ are numbers, (i.e., trees consisting of a leaf labeled by a number). Furthermore, only one of them is allowed to be an unknown so that the satisfiability of constraints can be tested using efficient techniques applicable to linear arithmetic.

Semantically, a program rule represents a set of potentially infinite number of ground rules obtained by replacing the variables in the rule with elements of the basic domain such that (1) the rule's constraints are satisfied, and (2) the rule's operations are well defined.

A program is a set of such rules and it provides the accumulated constraints obtained by successive rewritings of a query, using the abstract machine described in the section “Meta-Level Interpretation and Abstract Machine.”

In an implementation of Colmerauer's model, backtracking is initiated when: (1) the current set of constraints is unsatisfiable, or (2) it is meaningless to apply an operation to the given operands.

A fact deducible by erasure is a term (for instance, a query) which can be rewritten into the empty sequence using the program's rules.

A ground program rule:

\[ \text{head: } \text{body} \]

has two equivalent meanings: (1) the head can be rewritten as the body, or (2) if the elements of the body are found to be true (by erasure) then the head must also be true. Colmerauer's result embodies features of both propositions (a) and (b) of the previous section.

Given a program \( P \), its set of facts deducible by erasure is the smallest subset of the domain satisfying the logical implications represented by the rules of \( P \).

\*In [17] Colmerauer uses the term admissible element of the domain instead of deducible fact.

---

*The cognoscenti would equate CLP [] to Datalog without constant symbols.
Remarks
A comparison between the two approaches above is admittedly difficult. Although some of their authors' aims overlap, their approaches are different. Jaffar and Lassez are basically interested in a meta-theory applicable to any constraint language. Their emphasis is on the applicability of the theory to various domains rather than on the interaction between domains and the resulting complexity of an interpreter (or compiler) needed to process programs in the language. In contrast, Colmerauer’s theoretical work concentrates on providing the semantics of a family of languages in which great care is given to presenting a precise description of how the constructs pertaining to different domains should interact so as to achieve a feasible language processor.

In Colmerauer’s work the applicability of the theory to other domains is implicit and based on mathematical arguments. In Jaffar and Lassez’s approach the applicability is explicit and based on the existence of body of historical results in logic.

Domains and Constraint Satisfiability
The various individual domains that have been considered in the design of the main existing constraint languages will be addressed in this section. In the next section, the problems of combining several domains will be discussed.

Real Closed Fields (RCF)
This domain, as illustrated in the second and third examples in Table I, satisfies the conditions (1) and (2) of Jaffar and Lassez’s meta-theory. Tarski [45] has shown that there is a decision procedure for testing the satisfiability of systems of general equations and inequalities in that domain (also see [3]). However, the formidable complexity of the algorithm renders its direct usage by an interpreter infeasible. But this statement must be qualified: simplified versions of the algorithm may suffice for many interesting problems. Collins [12] has already implemented programs which can solve certain classes of problems in RCF.

More recently, algorithms for determining the so-called Gröbner Basis have been developed to test the satisfiability of systems of polynomial equations in RCF. The algorithm proposed by Buchberger [8] has been successfully used in implementing special constraint languages. However, a word of caution is in order. The Gröbner method tests whether a system of multivariate polynomial equations has a solution over the complex numbers. Therefore, its practical usage in a constraint language is to test the unsatisfiability of a system of polynomial equations. If the system fails the Gröbner test, then it has no complex solutions, and consequently, no real solutions. If the Gröbner test succeeds, further tests are needed to check if the solutions are real, and not complex.

The worst-case complexity of the Gröbner basis algorithm is doubly exponential (No “average case” complexity analysis is known). Note that it is possible to extend the Gröbner basis method to the case of inequalities by using Collins’ techniques [12].

Most of the existing constraint languages utilize subsets of RCF, which have efficient algorithms for testing the satisfiability of systems of constraints. Linear equations and inequalities are allowed in both CLP(R) and in Prolog III. The latter only deals with rational numbers, whereas the former includes reals. The design strategy for Prolog III is to consider the constraint satisfaction test as a “black box” which should always return a yes or no answer. By using multiple precision rational arithmetic, one can make an exact test for equality of numbers. On the other hand, CLP(R) utilizes real (floating point) numbers, so the test for equality must be done by making explicit the precision with which real numbers can be tested. This task is relegated to the programmer or to the language implementor (as with most current languages).

There is a wealth of algorithms and programs available for testing the satisfiability, and finding solutions, of systems of linear equations and inequalities. These algorithms have been developed in past decades by numerical analysts and specialists in operations research. Gaussian elimination and the simplex method [36] are typical examples of such algorithms, and they have been used by implementors of interpreters of constraint languages. It will be seen in the next section that these algorithms have to be adapted or modified to increase the efficiency of constraint language processors.

As mentioned previously, the predicate $\neq$ denoting the “disequality” of terms can also be introduced when using rational or real arithmetic; the equivalent predicate $d1f$ is already available in Prolog II, and its implementation in Prolog III is analogous to that of its predecessor [14]. Disequalities are not yet available in CLP(R): the user must simulate the effect by using other predicates.

An added advantage of this predicate is that it easily allows the use of the simplex method involving strict inequalities. A slack variable $X \geq 0$ combined with the constraint $X \neq 0$ produces the desired effect.

Booleans
The constraints in this case are formulas in two-valued boolean algebra using the standard operators ($\neg$, $\lor$, $\land$, $\equiv$, $\supset$). It is well known that the worst case complexity
of the algorithms for testing the satisfiability of a system of boolean equations is exponential with the number of variables. Nevertheless, little is known about the "average case" complexity. It is appropriate to draw a parallel between this situation and that encountered in the simplex method: although the worst case complexity of the simplex method is theoretically exponential, the average case is known to be polynomial.

There are several algorithms which have been proposed for testing the satisfiability of boolean formulas. The classic ones are described in most logic textbooks. The approach utilized in Prolog III is that of SL resolution, which has been successful in proving theorems in the predicate calculus [32]. (The reader is referred to [11] for a Prolog description of this method. The version of SL resolution used in Prolog III is that of Siegel [41]. A detailed description of the algorithms is provided in [4].

Constraint language designers should be aware of two characteristics of the SL resolution method. First, it requires that formulas be translated into clausal form, a task which may, in the worst case, also require exponential time and space. Second, in the case of satisfiability, the method can provide a simpler system of boolean formulas equivalent to the one being considered. In addition, the method is incremental in the sense that it avoids redoing computations. It will be seen in the next section that incrementality and simplification play an important role in the choice of algorithms for testing the satisfiability of systems of constraints.

Other methods have been recently proposed to test the satisfiability of systems of boolean equations. One of them, [9], makes extensive use of the exclusive-or operator; the other [39] avoids the translation into clausal form. The appropriateness of these methods in constraint language processors remains to be verified.

Lists
Although there are methods capable of checking the equality of two general lists containing variables, their complexity is forbidding [37]. This situation parallels that of Tarski's decision procedure for RCF. In Prolog programs the ubiquitous lists are considered as sequences of embedded appearances of the cons function. In Prolog III lists are given a special status as a domain. The key operation in this domain is concatenation. Its function is to append one list to another. Efficient implementation is guaranteed by requiring that the size of the first list be a known integer when the operation is performed. It can be shown that the concatenation of such strings can be done in constant time.

\[ S \cup R \cup (t_0 = t_0) \]

Design and implementation issues
A dilemma facing constraint language implementors is the selection of the appropriate algorithm for testing the satisfiability of systems of constraints in a given domain. Two extreme approaches can be envisioned. The first could be characterized as "killing a rabbit with an elephant gun," i.e., using an overly general and complex program to solve a relatively simple problem. (An example might be the use of the Gröbner Basis method to test the satisfiability of linear equations.) An implementor may initially frown at such wastage; nevertheless, this possibility should not be discarded a priori. The decreasing costs of (parallel) computers and the increasing costs of software development may encourage such "brute force" solutions.

The other extreme approach is to develop an arsenal of carefully designed program tools, each applicable to a specific type of constraint problem. This may be feasible when the complexity of determining the applicable tool is small, and the implementors have the resources needed to develop a variety of specialized programs.

A wise choice of an appropriate constraint algorithm will probably avoid these two extremes. Presently, it is fair to say that implementors lean more toward the second approach; however, this situation may change as machines become less expensive. It should be recalled that in the early constraint languages [5,43] a great deal of programming effort was devoted to replacing the solution of systems of linear equations by ad hoc methods. This is no longer true for the current constraint languages.

There is an important implementation consideration that appears to be fulfilled in both CLP(R) and Prolog III: the efficiency of processing Prolog programs (without constraints) should approach that of current Prolog interpreters, i.e., the overhead for recognizing more general constraints should be small. As indicated earlier, this strategy shows that, at least at present, the implementors prefer to avoid the "brute force" approach.

There are three factors that should be considered when selecting algorithms for testing the satisfiability of systems of constraints used in conjunction with CLP processors. They are:

- incrementality
- simplification
- canonical forms

The first is a desirable property which allows an increase in efficiency of multiple tests of satisfiability (by avoiding recomputations). This can be explained in terms of the abstract machine described earlier in this article: if the current system of constraints \( S \) is known to be satisfiable, the test of satisfiability of:

\[ S \cup R \cup (t_0 = t_0) \]
should be incremental, minimizing the computational effort required to check if the formula remains satisfiable or not. Classical Prolog interpreters have this property, since previously performed unifications are not recomputed at each inference step. There are modifications of Gaussian methods for solving linear equations which also satisfy this property. This is accomplished by introducing temporary variables and replacing the original system of equations by an equivalent one of the form (see [30]):

\[
\text{variable} = \text{linear terms involving only the temporary variables}
\]

The simplex method can also be modified to satisfy incrementality. Similarly, the SL resolution method for testing the satisfiability of boolean equations, and the Gröbner method for testing the satisfiability of polynomial equations, also have this property.

In nearly all the domains considered in the previous section, it may be possible to replace a set of constraints by a simpler set. This simplification can be time consuming, but is sometimes necessary. The implementor of CLP languages may have to make a difficult choice as to what level of simplification should occur at each step verifying constraint satisfaction. It may turn out that a system of constraints eventually becomes unsatisfiable, and all the work done in simplification is lost.

The problem of whether or not simplification should be performed is not unlike that of garbage collection (GC): one could call it a semantic GC. A well-known design strategy for classical GC is to perform it only when strictly necessary. A similar strategy might be used in the simplification of constraints: when a final result has to be output, it becomes essential to simplify it and present it to the reader in the clearest, most readable form.

An important function of simplification is to detect the assignment of a variable to a single value (e.g., from \( X \geq 1 \) and \( X \leq 1 \) one infers \( X = 1 \)). This property is essential when implementing the control procedure freeze(\( X, P \)), in which the procedure \( P \) is executed only when \( X \) is assigned to a single value.

The sorting example presented earlier illustrates well the problems involved in simplification. Recall that the result of a symbolic sorting of the list \( [X_1, X_2, X_3] \) is the set of constraints:

\[
X_1 \leq X_2 \text{ and } X_2 \leq X_3
\]

However, it is easy to check that the redundant constraint:

\[
X_1 \leq X_3
\]

would also be generated by the interpreter, and it should be eliminated by simplification.

The extent to which a final constraint should be simplified is a touchy design decision. In well-developed symbolic algebra languages, such as Macsyma, an arsenal of simplifying tools is available, and their selection is interactively made by the user. Perhaps the same will occur in future CLP languages.

A topic related to simplification is that of finding a minimal set of constraints, equivalent to a given set. In the case of booleans this minimal set can be defined and computed, but the computational costs are prohibitive (such algorithms are used in VLSI design however).

The incremental algorithms for testing the satisfiability of linear equations and inequalities, as well as that used in the Gröbner method for polynomial equations, are capable of discarding redundant equations; therefore, they perform some simplifications.

The canonical forms referred to earlier in this section can be viewed as (internal) representations of the constraints which facilitate both the tests of satisfiability and the ensuing simplifications. For example, in the case of the Gröbner method for solving polynomial equations, the input polynomials are internally represented in a normal form, such that variables are lexicographically ordered and the terms of the polynomials are ordered according to their degrees. This ordering is essential in performing the required computations. Also note that if two seemingly different constraints have the same canonical form, only one of them needs to be considered. Therefore, the choice of appropriate canonical forms deserves an important consideration in the implementation of CLP languages [34].

It is worth making a final comment about the eventual use of automatic theorem provers in testing the satisfiability of systems of constraints (see [6]). Just as in the case of simplification, the user may have to resort to several specialized tools capable of solving specific classes of theorems.

### A Comparison of Two Approaches

As noted earlier in this article, the Prolog III design philosophy for testing constraint satisfaction was to consider this operation as always capable of providing a yes-no answer, within a reasonable amount of time, and using present-day computers. To fulfill that goal, Colmerauer selected the domains of infinite trees, rationals, booleans and lists. The relations = and ≠ are applicable to all the domains (in the case of booleans = corresponds to = and ≠ to ≠). The allowed domains, relations, and operations for Prolog III are summarized in Table II. In contrast, the current CLP(R) processor covers only two domains: finite trees and reals. The predicate for expressing tree constraints is

---

*Reference [22] describes recent work in this area.

*Reals are implemented as floating point numbers.*
equality, and the constraint predicates applicable to reals are \(<\), \(\leq\), \(>\), \(\geq\).

Lists in CLP(R) are as in Prolog: a convenient syntactic notation is provided for representing them; however, it is up to the programmer to define explicit operations for their manipulation.

The strategy of the designers of CLP(R) is to allow a postponement of the satisfiability test of systems of constraints. It has been shown that, according to Tarski's results [45], these tests are theoretically possible; however, they are also prohibitively complex. The CLP(R) designers opted for a pragmatic approach: the test of satisfiability of non-linear equalities and inequalities is postponed with the expectation that they might become linear, in which case the known efficient algorithms are applicable. However, there is a caveat: if a solution is found in which the final set of constraints \(C\) includes non-linear constraints, it is up to the user to attempt to test whether they are satisfiable or not. Therefore the CLP(R) processor's output to such problems is : maybe, followed by \(C\). If the user can determine that \(C\) is indeed satisfiable, then the maybe answer is equivalent to a yes; otherwise, not only is the maybe a no, but it may become difficult to "manually" backtrack to the previous state of which a predecessor of \(C\) is indeed satisfiable.

The richness of domains in Prolog III, coupled with the design philosophy of considering the test for constraint satisfaction as a "black box" capable of providing a yes-no answer in reasonable time, forced its designer to introduce partial operations and to give careful consideration to how constraints in various domains can be combined. An example illustrates this situation: consider two linear inequalities \(I_1\) and \(I_2\) over the rationals. \(I_1 \lor I_2\) is not a valid boolean term in (standard) Prolog III\(^9\), since the test for its satisfiability is computationally complex. Note that a variant of the simplex method can be used to test the satisfiability of \(I_1\) or \(I_2\) individually. However, since \(I_1 \lor I_2\) may represent a non-convex polyhedron, the simplex method is no longer applicable.

A considerable part of the formal description of Prolog III is devoted to these problems [17]. At present, the type checking of programs is performed at compile time.

Two other interesting features of Prolog III are worth mentioning. The first is the existence of basic operators capable of determining the identifier representing the functor of a term and its arguments. In classic Prolog, this is done using ad hoc built-in procedures which are difficult to define formally. The second feature was also designed to replace the usage of the built-in procedure \(\text{call}(X)\) by the simpler term \(X\). Recall that in Prolog III, there is no distinction between terms and literals. The theory of rewriting rules (contained in the section "Colmerauer's Approach") is applicable to sequences of trees, and \(X\) is simply a tree.

\(^9\)Recent Prolog III implementations allow the delayed evaluation of such constraints.

### Table II. Allowed Domains, Relations, and Operations for Prolog III.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Relations and Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinite trees</td>
<td>([=, \neq])</td>
</tr>
<tr>
<td>rationals</td>
<td>Systems of linear equalities, inequalities and disequalities using the operations (+, -, \times).</td>
</tr>
<tr>
<td>booleans</td>
<td>([\neg, \land, \lor, \exists, =, \neq])</td>
</tr>
<tr>
<td>lists</td>
<td>Equalities and disequalities of lists. Concatenation, whose left operand is a list of known length. Transformation of lists into trees and vice versa.</td>
</tr>
</tbody>
</table>

### Applications Using CLP

Now that the main aspects of CLP languages and processors have been described, it is tempting to predict the various problem areas in which constraints will play an important role. Some of these areas are:

**Symbolic manipulation:** although Lisp and Prolog are currently the main languages in this area, it is quite probable that a CLP language may replace Prolog in the next few years. There is a close relationship between the aims of CLP and Macsyma-like languages. It is likely that some features of constraint languages such as inequalities, booleans, inversibility, etc., will be of interest to Macsyma users. On the other hand, developers of CLP languages will have a great deal to learn from the experience gained in implementing Macsyma.

**Numerical Analysis and Operations Research:** the proposed CLP languages allow their users to generate hundreds (or thousands) of equations and inequalities having special characteristics. For example, the generation of linear equations approximating Laplace's differential equations is easily done in a CLP language [24]. The same applies to Runge-Kutta methods for solving differential equations. The possibility of expressing inequalities in a computer language will likely attract the interest of specialists in operations research.

**Combinatorics:** non-deterministic languages like Prolog have been very successfully used in the solution of combinatorial problems. The availability of inequalities and disequalities \((\neq)\) will certainly extend the scope of problems which can easily be expressed by a program. A typical example is the now classic \(\text{SEND} + \text{MORE} = \text{MONEY}\) problem. Its solution using constraints is considerably
shorter and clearer than its counterpart which does not use them. It is also known that a version of the eight queens problem using the constraint \( \text{df}(\neq) \) is considerably more efficient than the classic one.

**Artificial Intelligence Applications:** the boolean constraints of Prolog III are already being utilized in the design of expert systems. It is expected that the possibility of expressing equations and inequations makes CLP languages attractive for writing natural language interfaces in which numeric values play an important role. The increased potential for invertibility makes CLP languages unique in programming certain applications. An example worth mentioning is that of option trading presented in [33].

**Engineering Applications:** The ease with which CLP can be used for generating large numbers of equations and inequations will have an impact in the solution of engineering problems. Two examples illustrate this type of application. Ohm's and Kirchhoff's laws can readily be used to generate equations describing the behavior of electrical circuits containing parallel and serial elements. The analysis of such circuits is easily and concisely done using CLP (see [24]). The second example is the detection of faulty components in electronic circuits using boolean constraints (see [17]).

**Conclusion**

As in most sciences, there has always been a valuable symbiosis among the theoretical and experimental practitioners of computer science including, of course, those working in logic programming. It is worth recalling the fundamental role played by the talented and creative "hackers" in our field. Three examples come to mind: the elimination of the \( \text{ocur} \) test in unifications; the \( \text{cut} \); and the \( \text{not} \) operator as defined in Prolog. These features were created by practical programmers and are here to stay. They provide a vast amount of food for thought for theoreticians. As mentioned earlier, the elimination of the \( \text{ocur} \) test was instrumental in the development of algorithms for unification of infinite trees. Although the concept of the \( \text{cut} \) has resisted repeated attempts for a clean semantic definition, its use is unavoidable in increasing the efficiency of programs. Finally, Prolog's \( \text{not} \) operator is now playing a key role in extending logic programs beyond Horn clauses.

The above comments remain valid in the area of CLP. The development of new practical algorithms for dealing with constraints will undoubtedly trigger a flurry of theoretical efforts to properly explain their behavior. Therefore, as in the case of Prolog, the increased availability of interpreters and compilers for CLP will encourage their practical usage and will provide the experience necessary to further refine the languages.

The remaining paragraphs will outline some of the promising research avenues that are wide open to researchers in CLP.

1. **Development of novel satisfiability algorithms in various domains:** it is safe to assume that linear equations and inequations (as well as disequalities) will be incorporated into most CLP languages. There is a substantial gap between the computational effort needed to solve such constraints and that needed to solve general polynomial equations and inequations [23]. It is worthwhile to study subsets of these constraints and to develop incremental algorithms for testing their satisfiability.

2. **Type theory:** since several domains may be combined in a single constraint language, it would be desirable to have some general means to test whether a given combination of domains and predicates should or should not be allowed in a program. The criterion in this case might be a reasonable execution time. (Some related results appear in [31]).

3. **Implementation:** a great deal of the success of Prolog is due to the development of a target language which can be efficiently used in executing programs. Warren's target language called Warren Abstract Machine (WAM) is far the most widely used lower-level code generated by current Prolog compilers [47]. One of its main advantages is that it replaces unification, whenever possible, by simpler sequences of tests and assignments. Although some of the implementations of existing constraint languages are WAM-based, it will still take a considerable effort to develop an efficient and well-accepted WAM model for CLP's handling multiple domains. Recent research [42] indicates that partial evaluation of CLP programs may yield substantial efficiency gains.

4. **Meta-level interpretation:** a substantial advantage offered by Prolog is that it allows a very concise description of its interpreter, using the language itself. It would be valuable to have the description of a full-fledged constraint language written in CLP (the author's group is presently developing a CLP interpreter written in Prolog). That interpreter would be very useful as a prototype for trying out the implementation of new features.

5. **Data-flow analysis:** it has already been mentioned that certain \( \text{cuts} \) could be eliminated by introducing constraints. Recall the example illustrating this possibility:

\[
\begin{align*}
p & : - q \quad 1 \quad \alpha. \\
p & : - \beta.
\end{align*}
\]

transformed into

\[
\begin{align*}
p & : - q \quad \alpha. \\
p & : - q' \quad \beta.
\end{align*}
\]
in which \( q \land q' = \text{false} \). It would be worthwhile to perform this computation automatically at compile time (note that a CLP interpreter may be capable of generating \( q' \) and testing if \( q \land q' \) is false). This could be accomplished by a data-flow analysis of the Cousot-Bruynooghe type \( [7, 18] \). Many other optimizations of CLP programs should be possible using these techniques.

6. **Parallelism:** this is probably one of the most exciting areas in CLP, especially since it relates to all the preceding topics.\(^{11}\) There already exist several parallel programs for solving equations, and for linear programming, which run on a variety of computer architectures (SIMD and MIMD machines).

A futuristic view of parallelism exploited for logic programming might envision a linkage between a loosely coupled shared-memory machine (the inference engine) and a connection machine dedicated to testing the satisfiability of systems of constraints (the unifier).

This author believes that CLP is one of the most promising and stimulating new areas in computer science. It amalgamates the knowledge and experience gained since the birth of computers in areas as varied as: numerical analysis, operations research, artificial languages, symbolic processing, AI, logic and mathematics. It represents a gold mine of concepts and problems which are bound to influence computer science in the coming decades.

**Acknowledgments.**

I am indebted to my colleague Timothy Hickey for sharing with me his insights on the underlying theory of CLP and for the numerous suggestions he made for improving the original manuscript. During my frequent trips to Marseilles, I have been fortunate to have had many discussions on CLP with Alain Colmerauer. In April of 1988, I had the privilege to organize a workshop on Languages and Constraints with Jean-Louis Lassez. A considerable part of the contents of this paper represent distilled thoughts and experiences gained from organizing that workshop. The original manuscript was meticulously read and commented upon by Alain Colmerauer, Jean-Louis Lassez, Joxan Jaffar, and Michael Maher. Practically all their suggestions were gladly taken into consideration in preparing the revised manuscript. Finally, I wish to express my sincere appreciation to Véronique Deschamps for her invaluable assistance in helping me refine the multiple versions of this work.

**References**


\(^{11}\)A recent dissertation \([40]\) deals with constraints and concurrent logic programming.


COMMUNICATIONS OF THE ACM/July 1990/Vol.33, No.7
41. Siegel, P. Representation et utilisation de la connaissance en calcul pro-
43. Steele, G.L. The implementation and definition of a computer program-

Computing Reviews (Monthly)

Editor-in-Chief: Aaron Finerman, University of Michigan, Ann Arbor, Michigan

Now in its 31st year, Computing Reviews is the only journal in the computer field devoted solely to providing reviews and critiques of the current professional literature. Each monthly issue contains reviews of information selected from more than 300 journals and thousands of books, proceedings, refereed articles, and conference proceedings.

Computing Reviews will help you to evaluate the quality and level of publications, monitor new developments quickly, reduce information and source overload, keep up to date on specific topics, locate citations of relevant topics and generate more effective bibliographies.

Each review is written by an expert in the field and includes a complete citation and consists entirely of the reviewer's opinions—which are independent of those of the editors and the publisher of Computing Reviews.

Computing Reviews (ISSN 0898-1984) is published monthly at a cost of $32.00 to members; $27.00 to student members; $115.00 to nonmembers.

Sample copies available upon request.

Circle # 18 on Reader Service Card

The First International Workshop on Interoperability in Multidatabase Systems will be held in Kyoto (the beautiful old capital city of Japan) along with ACM Int Conf. on Data Engineering. Original papers are solicited on topics including, but not limited to:

- Autonomy
- Resolving semantic heterogeneity
- Interdependence
- Federated & multi database tools
- Data definition
- New paradigms, object orientation
- Transactions
- Queries, updates, spec. & proc.
- Interoperability of DBMS & KBMS


Submit 7 copies of an extended abstract of at most 5 pages (t tro submit accepted) to Marek Rusilekiewicz, Houston U., C.S. Dept., Houston, TX 77204-0475, USA, (713) 749-4791, email: marek@cs.uh.edu or Yahiko Kambayashi, Kyushu U., CS-CE Dept., Hokozaki, Fukuoka 812, Japan, email: yahiko@ecs.kyushu-u.ac.jp, fax: 81-92-641-1825.

Program Committee: Y. Breitbart (USA), C. Cegzde (USA), T. Furukawa (Japan), H. Garcia-Molina (USA), G. Gottlob (Germany), S. Haider (USA), S. Hikita (Japan), L. Kershberg (USA), W. Lwin (France), E. Lévy (France), H. Makino (Japan), P. Musen (USA), G. Schlageter (Germany), A. Sheth (USA), H. Tirt (Finland), S. Uemura (Japan), R. Wang (USA), L. A. Vail (USA), M. Yoshikawa (Japan), E. Jiang (China), Z. Zhong; Publication: A. Sheth; Finance: R. Martin; Registration: A. Datta; Publicity: C. Perizzo.

Computing Reviews will help you to evaluate the quality and level of publications, monitor new developments quickly, reduce information and source overload, keep up to date on specific topics, locate citations of relevant topics and generate more effective bibliographies.

Each review is written by an expert in the field and includes a complete citation and consists entirely of the reviewer’s opinions—which are independent of those of the editors and the publisher of Computing Reviews.

Computing Reviews (ISSN 0898-1984) is published monthly at a cost of $32.00 to members; $27.00 to student members; $115.00 to nonmembers.

Sample copies available upon request.