TR-364

CAL : A Theoretical Background of Constraint Logic Programming and its Applications

by
K. Sakai & A. Aiba

April, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

CAL: A Theoretical Background of Constraint Logic Programming and Its Applications

KŌ SAKAI and AKIRA AIBA
Institute for New Generation Computer Technology
21F, Mita Kokusai Building,
4-18, Mita 1-Chome, Minato-ku, Tokyo 108, Japan
sakai%icot.jp@relay.cs.net
{enea,inria,kddlab,mit-eddie,ukc}!icot!sakai

April 11, 1988

Abstract

Constraint logic programming (CLP) is an extension of logic programming by introducing the facility of writing and solving constraints in a certain domain. CAL (Contrainte avec Logique) is a CLP language in which (possibly non-linear) polynomial equations can be written as constraints, while almost all the other CLP languages proposed so far have concentrated only on linear equations and inequations. This paper describes a general semantics of CLP including CAL, and shows the validity of CAL in this framework.

1 Introduction

A paradigm called *Constraint Logic Programming* (CLP) was proposed by Colmerauer [2], and Jaffar and Lassez [7]. A similar paradigm (or language) was proposed by Dincbas, Simonis, and van Hentenryck [3]. Programs written in logic programming languages like Prolog are executed by unification. CLP is an attempt to increase the descriptive power of logic programming by employing constraint solving instead of unification as its execution mechanism. In this sense, constraint solving can be viewed as a generalization of unification.

The idea of programming by constraints is not new, e.g. [4] and [12]. However, it is in the framework of logic programming that constraints give full play to their ability. There are many advantages in the combination of logic programming and constraints. The most outstanding feature of constraint programming is that it allows the declarative description of problems. This feature should be preserved when controls are described as well. Declarative description of problems is also a feature of logic programming and, therefore, is inherited by the combination naturally.

In fact, there is a simple and unified framework for the declarative and operational semantics of CLP. This may not be true for a language in which controls are described operationally. A simple generalization of the ordinary goal-reduction technique of logic programming can be viewed as the operational semantics of CLP.

Traditional logic programming possesses logical, functional, and operational semantics, which coincide with each other [8], [10], and [15]. Jaffar and Lassez showed that CLP is a generalization of traditional logic programming in the sense that it possesses these three semantics [7]. In addition, they introduced algebraic semantics of CLP. According to [7], execution steps of CLP programs depend upon decision of the satisfiability of constraints in a given domain. However, we require more; the canonical forms of constraints should be computed if the constraints are satisfiable. This is a very similar situation to that in ordinary logic programming where the unification procedure decides the satisfiability of equations in the Herbrand universe and computes the most general unifier if satisfiable. Since equations can be considered constraints, the operational model of CLP is an extension of the model (of usual logic programming) based on unification. Thus, we are almost on the side of Jaffar and Lassez in the theoretical argument, but are on a different side in the details.

This paper describes the theoretical foundation, implementation, and application of CAL (Contrainte avec Logique), which is the CLP language we are developing. After a preliminary definition of the logical semantics of CLP in Section 2, Section 3 presents functional semantics and Section 4 presents operational semantics. Section 5 discusses the canonical forms of constraints which are appropriate as the answers from the system. The language, CAL, which treats polynomial equations as constraints is introduced in Section 6, and Boolean CAL, which is a version of CAL that treats Boolean equations as constraints, is described in Section 7.

2 CLP on Many Sorted Algebra

This section presents basic notions to describe the semantics of CLP. Let S be a finite set of sorts, F a set of function symbols, C a set of constraint symbols, P a set of predicate symbols, and V a set of variables. A sort is assigned to each variable and function symbol. A finite (possibly empty) sequence of sorts, called signature, is assigned to each function, predicate, and constraint symbol. We write $v : s$, $f : s_1 s_2 \ldots s_n \to s$, and $p : s_1 s_2 \ldots s_n$ if a variable, v, has a sort, s, if a function symbol, f, has a signature, $s_1 s_2 \ldots s_n$, and a sort, s, and if a predicate or constraint symbol, p, has a signature, $s_1 s_2 \ldots s_n$, respectively.

Terms and their sorts are defined inductively as follows.

1. A variable of sort s is a term of sort s.
2. If f is a function symbol such that $f : s_1 s_2 \ldots s_n \to s$, and $t_1, t_2, \ldots, t_n$ are terms of sorts $s_1, s_2, \ldots, s_n$ respectively, then $f(t_1, t_2, \ldots, t_n)$ is a term of sort s.

Atomic formulae and an atomic constraints are defined as follows.

3. If p is a predicate symbol such that $p : s_1 s_2 \ldots s_n$, and $t_1, t_2, \ldots, t_n$ are terms of sorts $s_1, s_2, \ldots, s_n$ respectively, then $p(t_1, t_2, \ldots, t_n)$ is an atomic formula.
4. If c is a constraint symbol such that $c : s_1 s_2 \ldots s_n$, and $t_1, t_2, \ldots, t_n$ are terms of sorts $s_1, s_2, \ldots, s_n$ respectively, then $c(t_1, t_2, \ldots, t_n)$ is an atomic constraint.

We write $t : s$ if a term t has a sort s. The set of terms, atomic formulae, and atomic constraints are denoted by $T(F, V)$, $A(P, F, V)$, and $A(C, F, V)$, respectively. A constraint is a finite (possibly empty) set of atomic constraints. Intuitively, a constraint is a finite conjunction of atomic constraints. The empty constraint means true.

We assume that for each sort, s, there is a special constraint symbol, $=_s$, of signature ss. For this symbol, we use infix notation, and the suffix s may be omitted if there is no danger of confusion.

A combination D of a class of sets, $\{D(s) | s \in S\}$, a class of functions, $\{D(f) | f \in F\}$, and a class of functions, $\{D(c) | c \in C\}$, satisfying the following conditions is called a *structure*. A structure plays the same role as the Herbrand universe does in the semantics of ordinary Prolog.

1. If f is a function symbol such that $f : s_1 s_2 \dots s_n \rightarrow s$, then $D(f)$ is a function from $D(s_1) \times D(s_2) \times \cdots \times D(s_n)$ to $D(s)$.

2. If c is a constraint symbol such that $c : s_1 s_2 \dots s_n$, then $D(c)$ is a function from $D(s_1) \times D(s_2) \times \cdots \times D(s_n)$ to $\{\mathbf{false}, \mathbf{true}\}$.

In what follows, let D be a fixed structure. Suppose that $D(=_s)$, which is a function from $D(s) \times D(s)$ to $\{\mathbf{false}, \mathbf{true}\}$, satisfies the following condition.

$$D(=_s)(x, y) = \textit{if } x = y \textit{ then } \mathbf{true} \;\; \textit{else } \mathbf{false}$$

Note that $=_s$ here plays the same role as unification in ordinary Prolog.

A class, I, of functions, $\{I(p) | p \in P\}$, satisfying the following conditions is called an *interpretation*, which plays the same role as an Herbrand interpretation in the semantics of ordinary Prolog.

3. If p is a predicate symbol such that $p : s_1 s_2 \dots s_n$, then $I(p)$ is a function from $D(s_1) \times D(s_2) \times \cdots \times D(s_n)$ to $\{\mathbf{false}, \mathbf{true}\}$.

   An assignment is a function, Θ, from V to $\bigcup_s D(s)$ satisfying the following condition.

4. If $v : s$, then $v\Theta \in D(s)$. (We use the symbol, Θ, in postfix notation as usual.)

An assignment, Θ, can be naturally extended to a function of $T(F, V)$ and $A(C, F, V)$. Then $t\Theta \in D(s)$ if t is a term of sort s, and $p\Theta$ is $\mathbf{false}$ or $\mathbf{true}$ if p is an atomic constraint. Let C be a constraint. If there exists an assignment, Θ, such that $c\Theta = \mathbf{true}$ for every $c \in C$, then C is said to be *satisfiable*, and Θ is called a *solution* of C. Similarly, Θ can be extended to a function of $A(P, F, V)$ into $\{\mathbf{false}, \mathbf{true}\}$, denoted ΘI, if an interpretation, I, is given.

A *program clause*, which is an extension of a definite clause, is an expression in the form of $p : -p_1, p_2, \dots, p_n$ $(n \geq 0)$, where p is an atomic formula and each p_i is an atomic constraint or an atomic formula. A finite set of program clauses is called a (constraint logic) program. Let L be a program. An interpretation is called a model of L if for any program clause $(p : -p_1, p_2, \dots, p_n) \in L$, and for any assignment, Θ, $p_1\Theta I = p_2\Theta I = \cdots = p_n\Theta I = \mathbf{true}$ implies $p\Theta I = \mathbf{true}$.

3 Functional Interpretation of a Program

First, we extend the function given by van Emden and Kowalski [15] for CLP. Let there be a program, L. Based on an interpretation, I, we can define another interpretation, J, as follows.

```
$J(p)(d_1, d_2, \dots, d_n) =$
    *if* there is a program clause $p(t_1, t_2, \dots, t_n) : -p_1, p_2, \dots, p_m \in L$ and an assignment, $\Theta$,
      such that $p_1\Theta I = p_2\Theta I = \dots = p_m\Theta I = \mathbf{true}$ and $d_1 = t_1\Theta, d_2 = t_2\Theta, \dots, d_n = t_n\Theta$
    *then* true
    *else* false
```

Since interpretation J is dependent on program L and interpretation I, let us denote it $T(L, I)$. Then $T(L, \_)$ forms a function which maps one interpretation to another. An interpretation, I, is said to be *less* than another interpretation, J, denoted $I \leq J$, if the following hold. For every predicate symbol $p : s_1 s_2 \dots s_n$, and for every element $d_1 \in D(s_1), d_2 \in D(s_2), \dots, d_n \in D(s_n)$, if $I(p)(d_1, d_2, \dots, d_n) = \mathbf{true}$, then $J(p)(d_1, d_2, \dots, d_n) = \mathbf{true}$. Proof of the following proposition is a routine.

Proposition 3.1 *The set of all the interpretations forms a complete lattice with respect to $\leq$, and $T(L,\_)$ is continuous on it. That is to say, the following conditions hold.*

1. *If $I \leq J$ then $T(L,I) \leq T(L,J)$.*
2. *If $I_1 \leq I_2 \leq \ldots$, then $\sup\ T(L,I_i) = T(L, \sup\ I_i)$.*

For any ordinal number, α, interpretations $T \uparrow \alpha$ and $T \downarrow \alpha$ are defined by transfinite induction as follows.

$$\begin{array}{rcl} T \uparrow \alpha & = & \textit{if } \alpha \text{ is a successor ordinal, } \beta+1, \textit{ then } T(L, T\uparrow\beta) \textit{ else } \sup\{T\uparrow\beta \mid \beta < \alpha\} \\ T \downarrow \alpha & = & \textit{if } \alpha \text{ is a successor ordinal, } \beta+1, \textit{ then } T(L, T\downarrow\beta) \textit{ else } \inf\{T\downarrow\beta \mid \beta < \alpha\} \end{array}$$

The definition after *else* is adopted also when $\alpha = 0$. Thus, $T \uparrow 0$ becomes the least element with respect to $\leq$. That is to say, for every predicate symbol $p : s_1 s_2 \ldots s_n$, and for every element, $d_1 \in D(s_1), d_2 \in D(s_2), \ldots, d_n \in D(s_n)$, $T \uparrow 0(p)(d_1, d_2, \ldots, d_n) = \texttt{false}$. On the other hand, $T \downarrow 0$ becomes the greatest element with respect to $\leq$. That is, for every predicate symbol, $p : s_1, s_2, \ldots, s_n$, and for every element, $d_1 \in D(s_1), d_2 \in D(s_2), \ldots, d_n \in D(s_n)$, $T \downarrow 0(p)(d_1, d_2, \ldots, d_n) = \texttt{true}$.

It is easy to show the following.

$$\begin{array}{c} T\uparrow 0 \leq T\uparrow 1 \leq T\uparrow 2 \leq \ldots \\ T\downarrow 0 \geq T\downarrow 1 \geq T\downarrow 2 \geq \ldots \end{array}$$

From Proposition 3.1 (1) and the fixed-point theorem with respect to order homomorphisms of a complete lattice, $T(L,\_)$ has the least and the greatest fixed-points. We write them $\mathrm{lfp}(T,L)$ and $\mathrm{gfp}(T,L)$, respectively. Then, for some sufficiently large ordinals, α and β, $\mathrm{lfp}(P,T) = T\uparrow\alpha$ and $\mathrm{gfp}(T,L) = T\downarrow\beta$. In fact, it is easy to show that $\mathrm{lfp}(T,L) = T\uparrow\omega$ from Proposition 3.1 (2). In general, the greatest fixed-point $\mathrm{gfp}(T,L)$ is different from $T\downarrow\omega$.

Lemma 3.1 *For any program, L, the following conditions hold.*

1. *$T(L,I) \leq I$ if and only if I is a model of L. Especially, the greatest element, $T\downarrow 0$, is the greatest model of L.*
2. *$\mathrm{lfp}(T,L)$ is a model, and for any model, I, $\mathrm{lfp}(T,L) \leq I$. Therefore, $\mathrm{lfp}(T,L)$ is the least model of L.*

Here, we define the syntactical counterpart to the function, $T(L,\_)$. Consider a pair of an atomic formula, p, and a satisfiable constraint, C. For convenience, we denote this pair $p : -C$ and call it a *QA-pair* (question and answer). We denote the set of all QA-pairs QA. From a subset, S, of QA, another subset, T, is defined as the set of all QA-pairs, $\{p(s_1, s_2, \ldots, s_n) : -C\}$, such that there is a program clause, $p(t_1, t_2, \ldots, t_n) : -p_1, p_2, \ldots, p_m \in L$, and

1. For each p_i, p_i is an atomic formula such that $(p_i : -C_i) \in S$, or an atomic constraint such that $C_i = \{p_i\}$,
2. $C = \{s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n\} \cup C_1 \cup C_2 \cup \ldots \cup C_m$,
3. C is satisfiable.

We denote T, defined above, $Q(L,S)$. Then $Q(L,\_)$ is a function which maps one subset of QA to another. Function $Q(L,\_)$ has a similar property to $T(L,\_)$ with respect to the inclusion relation of sets $\subseteq$.

Proposition 3.2 *$Q(L,\_)$ is continuous with respect to the inclusion relation of sets $\subseteq$. That is, the following conditions hold.*

1. *If $S \subseteq T$, then $Q(L,S) \subseteq Q(L,T)$.*
2. *If $S_1 \subseteq S_2 \subseteq \ldots$, then $\bigcup Q(L,S_i) = Q(L, \bigcup S_i)$.*

Similarly, $Q \uparrow \alpha$, and $Q \downarrow \alpha$ are defined as follows.

$$\begin{aligned} Q \uparrow \alpha &= \textit{if } \alpha \text{ is a successor ordinal, } \beta + 1, \textit{ then } Q(L, Q \uparrow \beta) \textit{ else } \bigcup\{Q \uparrow \beta \mid \beta < \alpha\} \\ Q \downarrow \alpha &= \textit{if } \alpha \text{ is a successor ordinal, } \beta + 1, \textit{ then } Q(L, Q \downarrow \beta) \textit{ else } \bigcap\{Q \downarrow \beta \mid \beta < \alpha\} \end{aligned}$$

In particular, $Q \uparrow 0 = \emptyset$ and $Q \downarrow 0 =$ **QA**. The following are also routines.

$$\begin{gathered} Q \uparrow 0 \subseteq Q \uparrow 1 \subseteq Q \uparrow 2 \subseteq \cdots \\ Q \downarrow 0 \supseteq Q \downarrow 1 \supseteq Q \downarrow 2 \supseteq \cdots \end{gathered}$$

$Q(L,\_)$ has the least fixed-point, $\text{lfp}(Q,L)$, and the greatest fixed-point, $\text{gfp}(Q,L)$. For sufficiently large ordinals, α and β, $\text{lfp}(Q,L) = Q \uparrow \alpha$, and $\text{gfp}(Q,L) = Q \downarrow \beta$. In fact, $\text{lfp}(Q,L) = Q \uparrow \omega$, but $\text{gfp}(Q,L)$ is different from $Q \downarrow \omega$, in general.

For $S \subseteq$QA, an interpretation, $|S|$, is defined as follows.

$$\begin{aligned} |S|(p)(d_1, d_2, \ldots, d_n) = & \\ & \textit{if} \text{ there is a QA-pair } (p(t_1, t_2, \ldots, t_n) : -C) \subseteq S \text{ and an assignment, } \Theta I, \\ & \quad \text{such that } d_1 = t_1\Theta, d_2 = t_2\Theta, \ldots, d_n = t_n\Theta \text{ and } \Theta \text{ is a solution of } C \\ & \textit{then } \textbf{true} \\ & \textit{else } \text{ false} \end{aligned}$$

Lemma 3.2 *For any program, L, and for any ordinal, α, $T \uparrow \alpha = |Q \uparrow \alpha|$ and $T \downarrow \alpha = |Q \downarrow \alpha|$.*

By the above lemma, $\text{lfp}(T,L) = |\text{lfp}(Q,L)|$ and $\text{gfp}(T,L) = |\text{gfp}(Q,L)|$.

4 Operational Interpretation of Programs

This section defines an operational model for CLP. A formula in the form of $p_1, p_2, \ldots, p_n; C$ is called a goal, where each p_i is an atomic constraint or an atomic formula, and C is a satisfiable constraint. A satisfiable constraint is called a *successful* goal, when it is viewed as a goal such that $n = 0$. Let L be a program. The (extended) *SLD-resolution* is the process which obtains a new goal from another goal $p_1, p_2, \ldots, p_n; C$ in the following way.

1. If p_1 is an atomic constraint such that $D = \{p_1\} \cup C$ is satisfiable, then the goal, $p_2, \ldots, p_n; D$, is obtained.

2. If $p_1 = p(s_1, s_2, \ldots, s_m)$ is an atomic formula such that there is a program clause $(p(t_1, t_2, \ldots, t_m) : -q_1, q_2, \ldots, q_k) \in P$ such that $D = \{s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m\} \cup C$ is satisfiable, then the goal, $q_1, q_2, \ldots, q_k, p_2, \ldots, p_n; D$, is obtained.

A sequence of goals, $G_0, G_1, \ldots, G_n$, is called an *SLD-resolution sequence* if each G_{i+1} is obtained from G_i by SLD-resolution. Here, we define a success set, $SS(L)$.

$SS(L) = \{\ (p : -C) \in \mathrm{QA}\ |$
there exists an SLD-resolution sequence which begins with the goal, $p; \emptyset$,
and ends with the successful goal, $C\}$.

Theorem 4.1 *For any program, L,* $|\mathrm{lfp}(Q, L)| = |SS(L)|$.

The reader can easily see that, if p is input as a query, a constraint, C, such that $(p : -C) \in SS(L)$ is output as an answer from the system. The above theorem guarantees the correctness of this mechanism.

5 Constraint Solving and Canonical Forms

According to the operational model of CLP described in the previous section, decision of the satisfiability of constraints is necessary and sufficient to execute a program by (extended) SLD-resolution. However, a satisfiable constraint, as it is, may not be satisfactory as output from the system if it is assured to be only satisfiable. For example, the constraint, $\{x+y = 3,\ x-y = 1\}$, is satisfiable, and is therefore qualified to be output as an answer according to the definition in the previous section. It is the answer $\{x = 2,\ y = 1\}$, however, that users actually want in many cases. In this sense, *constraint solving* should not be a mere decision of the satisfiability of constraints but conversion of constraints into another form that users can understand easily.

Two constraints are said to be equivalent if they have the same solutions. We write $C \sim D$ if C and D are equivalent. For example, $\{x + y = 3,\ x - y = 1\} \sim \{x = 2,\ y = 1\}$. Clearly, $\sim$ defines an equivalence relation for constraints. Suppose that for each equivalence class, E, there is a representative, $E\downarrow$. The equivalence class to which C belongs is denoted $[C]$, and the representative, $[C]\downarrow$, is called the canonical form of C. Let us call an algorithm, A, satisfying the following conditions, a *constraint solver* with respect to $\downarrow$.

1. A decides the satisfiability of an arbitrary constraint.

2. A computes the canonical form of an arbitrary satisfiable constraint.

When there is a constraint solver, as defined above, the SLD-resolution in the previous section can be improved; it computes the canonical form of the union, D, of constraints instead of merely making the union. Actually, unification of ordinary logic programming can be seen as computation of the canonical form of equality constraints in the Herbrand universe. Moreover, computation of the canonical forms may make program execution more efficient, if there is an algorithm that solves constraints incrementally based on the canonical forms.

6 CAL (Contrainte avec Logique)

A language named CLP(R) was developed at Monash University as an instance of CLP languages [9] and [5]. In CLP(R), constraints in the form of linear equations and linear inequations can be handled. There is another important CLP language: Prolog III of Colmerauer [2]. In Prolog III, linear constraints over rational numbers and Boolean constraints can be handled. This section describes our CLP language, *CAL (Contrainte avec Logique)*. The main feature of CAL is that it has the facility of handling constraints in the form of (possibly non-linear) polynomial equations.

6.1 Language and Domain

The language of CAL is defined as follows.

$$\begin{aligned}
&S = \{\text{AN}\}\\
&F = \{\times, +\} \cup \{\text{fraction}\}\\
&C = \{=\}\\
&P = \{\text{string of alphanumeric characters starting with a lowercase letter}\}\\
&V = \{\text{string of alphanumeric characters starting with an uppercase letter}\}
\end{aligned}$$

In the actual CAL system, there is a sort of Herbrand universe for a compatibility with Prolog. Here, however, we assume that there is only one sort AN of algebraic number for simplicity. If there is only one sort, the sort of each symbol need not be specified, and each signature is determined only by arity.

We define a structure for the above language as follows.

$$\begin{aligned}
D(\text{AN}) =&\ \text{the set of all algebraic numbers}\\
D(\times) =&\ \text{multiplication}\\
D(+) =&\ \text{addition}\\
D(\text{fraction}) =&\ \text{the rational number it denotes}
\end{aligned}$$

It is clear that we can write polynomial equations as constraints.

6.2 Constraint Solver: Buchberger Algorithm and Gröbner Bases

Buchberger introduced the notion of Gröbner bases and devised an algorithm to compute the Gröbner base of a given finite set of polynomials [1]. This algorithm has been widely used in the field of computer algebra over the past few years. Gröbner bases satisfy the conditions which are listed in Section 5 almost perfectly. Therefore, the CAL interpreter utilized the Buchberger algorithm as the constraint solver. First of all, we describe the theoretical background of Gröbner bases and the Buchberger algorithm.

Without loss of generality, we can assume that all polynomial equations are in the form of $p = 0$. Let $E = \{p_1 = 0, \ldots, p_n = 0\}$ be a system of polynomial equations, and I the ideal in the ring of all the polynomials generated by $\{p_1, \ldots, p_n\}$. The following close relation between the elements of I and the solutions of E is well known as the Hilbert zero point theorem [6].

Theorem 6.1 *Let p be a polynomial. Every solution of E is also a solution of $p = 0$, if and only if there exists a natural number n such that p^n is an element of I.*

Moreover, the following corollary is important to determine the satisfiability of constraints.

Corollary 6.1 *E has no solution if and only if $1 \in I$.*

Thus, the problem of solving constraints is reduced to the problem of determining whether a polynomial belongs to the generated ideal. Buchberger gave an algorithm to determine whether a polynomial belongs to the ideal. A rough sketch of the algorithm is as follows (see [1] for a precise definition).

Let there be a certain ordering among monomials and let a system of polynomial equations be given. An equation can be considered a rewrite rule which rewrites the greatest monomial in the equation to the polynomial consisting of the remaining monomials. For example, if the ordering is lexicographic, a polynomial equation, $Z - X + B = A$, can be considered as a rewrite rule, $Z \rightarrow X - B + A$. Two rewrite rules whose left hand sides are not mutually prime are said to *overlap*. In this case, the least common multiple (LCM) of their left hand sides can be rewritten in two ways by these two rules, which may produce different results. The resulting pair is called a *critical pair*. If further rewriting does not succeed in converging a critical pair, the pair is said to be *divergent* and is added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting system. The confluent rewriting system thus obtained is called a *Gröbner base* of the original system of equations. The following theorem establishes the relationship between ideals and Gröbner bases.

Theorem 6.2 *Let R be a Gröbner base of a system of equations $\{p_1 = 0, \ldots, p_n = 0\}$, and let I be an ideal generated by $\{p_1, \ldots, p_n\}$. A polynomial, p, belongs to I if and only if p is rewritten to 0 by R.*

Moreover, the following theorem guarantees the validity of considering the reduced Gröbner bases as the canonical forms of constraints. A Gröbner base is said to be *reduced* if it has no two rules, one of which rewrites the other.

Theorem 6.3 *Suppose that the ordering among monomials is fixed. Let E and F be systems of equations. Then if the ideal generated from E is the same as that from F, then the reduced Gröbner base of E is same as that of F.*

Since the relation between the solutions and the ideal described in theorem 6.1 is incomplete, the reduced Gröbner bases do not satisfy the requirements in Section 5 completely. For instance, constraints $\{X = 0\}$ and $\{X^2 = 0\}$ have exactly the same solutions. However, the reduced Gröbner bases are different. That is, that of the first constraint is $\{X \to 0\}$, while that of the second is $\{X^2 \to 0\}$. Namely, the Gröbner base of the radical of the generated ideal, I, i.e. $\{p|p^n \in I\}$, is more desirable than that of ideal I itself for the purpose of the CAL system. Therefore, we are looking for an algorithm which computes the Gröbner bases of the radical. However, we do not think that such an algorithm is critical, because the ordinary Gröbner bases seem to work satisfactorily.

6.3 Program Example

First, we will illustrate the execution of a CAL program by an example. As explained in the previous section, CAL can distinguish itself when constraints are non-linear. The following is an example of proving a geometrical theorem; the four midpoints of the edges of a quadrangle form a parallelogram. The program is as follows.

```
mid(AX,AY,BX,BY,CX,CY) :- AX+CX=2*BX, AY+CY=2*BY.
para(AX,AY,BX,BY,CX,CY,DX,DY) :- (AX-BX)*(CY-DY) == (AY-BY)*(CX-DX).
```

The above clauses state the conditions for midpoint and parallel. The mid clause states that point (BX,BY) is a midpoint of segment (AX,AY)-(CX,CY). The para clause checks whether segment (AX,AY)-(BX,BY) and segment (CX,CY)-(DX,DY) are parallel. In this clause, for convenience, we used a meta-predicate which does not fit the purely logical framework, namely, the predicate ==. This predicate checks the equality of its right and left hand sides under the current collection of constraints, just as the same predicate symbol does in Prolog. This kind of control seems to be indispensable to write an actual application program in any language. The body of the para clause is obtained by transforming the equation:

$$\frac{\mathtt{AY} - \mathtt{BY}}{\mathtt{AX} - \mathtt{BX}} == \frac{\mathtt{CY} - \mathtt{DY}}{\mathtt{CX} - \mathtt{DX}}$$

representing the equality of the tangents of the two segments.

To prove the above problem by this program, the following goal sequence should be evaluated.

```
?-  mid(0,0,x4,y4,x1,y1),
    mid(x1,y1,x5,y5,x2,y2),
    mid(x2,y2,x6,y6,x3,0),
    mid(x3,0,x7,0,0,0),
    para(x4,y4,x5,y5,x7,0,x6,y6),
    para(x4,y4,x7,0,x5,y5,x6,y6).
```

Refer to the following figure for the coordinates.

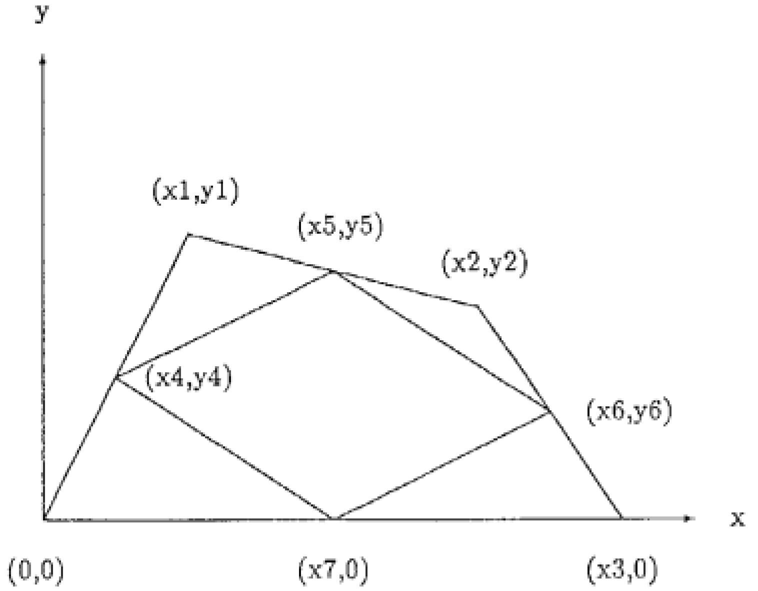

Figure 1 Coordinates for a Geometric Problem

The above goal is evaluated as follows.

1. The first predicate, `mid(0,0,x4,y4,x1,y1)`, is matched with the head of the `mid` clause. Then constraints `0+x1=2*x4` and `0+y1=2*y4` are obtained and their Gröbner base, $\{x4 \rightarrow 1/2 * x1, y4 \rightarrow 1/2 * y1\}$, is computed.

2. Similarly the next three predicates are matched with the head of the same clause, and constraints `x1+x2=2*x5`, `y1+y2=2*y5`, `x2+x3=2*x6`, `y2+0=2*y6`, `x3+0=2*x7`, and `0+0=2*0` are obtained. A new Gröbner base is always computed incrementally from an old base, when each constraint is obtained.

3. The next predicate, para(x4,y4,x5,y5,x7,0,x6,y6), is matched with the head of the para clause. Then the equation, (x4-x5)*(0-y6) == (y4-y5)*(x7-x6), is checked under the constraints obtained so far. Both sides of this equation are simplified to x2*y2 by the current Gröbner base. Therefore, the equation holds under the constraints.

4. The last predicate, para(x4,y4,x7,0,x5,y5,x6,y6), is processed similarly, and the equation, (x4-x7)*(y5-y6) == (y4-y5)*(x7-x6), is checked. Both sides of this equation are simplified to y1*x1-y1*x3), and therefore, the equation holds.

As explained in (2), during execution of CAL programs, each encounter with an atomic constraint causes invocation of the constraint solver. If the new constraint is proved to be inconsistent with the previous ones, the execution fails and backtracks.

The above example uses Gröbner bases indirectly via the predicate, ==. The following is an example of using Gröbner bases directly.

```
sur(H,A,S) :- A*H=2*S.
right(A,B,C) :- A^2+B^2=C^2.
tri(A,B,C,S) :- C=CA+CB, right(CA,H,A), right(CB,H,B), sur(H,C,S).
```

where A^2 is a syntax sugar of A*A, and so are the others.

The first predicate expresses the formula to compute the area of a triangle from its height and baseline length. The second is the Pythagorean theorem. The third asserts that every triangle can be divided into two right-angled triangles. (See Figure 2.)

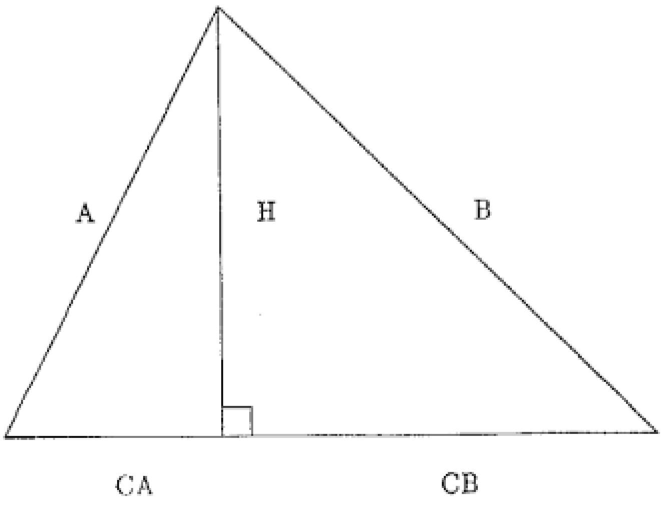

Figure 2 Area of a Triangle

If the goal, tri(A,B,C,S), in which all the parameters are free, is given, this program computes a Gröbner base consisting of seven rules. An outstanding feature of this base is that it includes a formula constructed by variables A, B, C, and S only, namely:

```
S^2=(-A^4-B^4-C^4+2*B^2*C^2+2*C^2*A^2+A^2*B^2)/16
```

which is the famous Heron's formula in developed form. Of course, this program can be executed by a goal with concrete parameters. For example, when the goal `tri(3,4,5,S)` is given, the program answers that `S^2=36`.

7 Boolean CAL

CAL described in the previous section is for constraints in the form of polynomial equations over algebraic numbers. We also implemented another version of CAL, in which Boolean equations can be written as constraints. A typical domain for this version of CAL is the set of truth values. This constraint solver employed a similar algorithm to Buchberger's but was modified for Boolean constraints [11].

In Boolean CAL, we can write programs which need logical evaluation very easily and naturally. For instance, it is an easy task to write a program which verifies the correctness of logical circuits.

7.1 Language and Domain

First, let us define the language and the structure of Boolean CAL as follows.

$$\begin{aligned}
S &= \{\mathrm{BA}\}\\
F &= \{\wedge, \oplus, \perp, \top\}\\
C &= \{=\}\\
P &= \{\text{string of alphanumeric characters starting with a lowercase letter}\}\\
V &= \{\text{string of alphanumeric characters starting with an uppercase letter}\}
\end{aligned}$$

$$\begin{aligned}
D(\mathrm{BA}) &= \text{ an arbitrary Boolean algebra}\\
D(\wedge) &= \text{ conjunction}\\
D(\oplus) &= \text{ exclusive disjunction}\\
D(\perp) &= \text{ false}\\
D(\top) &= \text{ true}
\end{aligned}$$

In the actual system, other logical connectives such as disjunction, implication, and negation are also included in F. However, since it is well known that they can be defined from $\wedge$, $\oplus$, $\perp$, and $\top$, we have omitted them for simplicity.

7.2 Boolean Gröbner Bases

There are many known procedures to decide the satisfiability of Boolean equations. Of these procedures, the semantic unification method is one of the most promising. For instance, Dincbas employed it as a constraint solver for his language [3].

However, a Gröbner base type approach can be applied to the Boolean equations as well as to ordinary algebraic equations. This approach is better than the semantic unification method in the following points.

1. It is not necessary to introduce extra variables which are not explicitly written in the program or the goal. Thus, output from the system is easy for the user to understand.

2. Every constraint has its canonical form in the sense of Section 5, and the canonical form is computed efficiently.

There is an algorithm to compute a Boolean Gröbner base of a given Boolean constraint [11]. Here, we summarize several important properties of Boolean Gröbner bases. Let E be a system of Boolean equations, p a Boolean polynomial, I the ideal generated by E, and R a Boolean Gröbner base of E. The following is the Boolean counterpart to the Hilbert zero point theorem.

Theorem 7.1 *Every solution of E is also a solution of $p = 0$ if and only if $p \in I$.*

Corollary 7.1 *E has no solutions if and only if $1 \in I$.*

Theorem 7.2 *p is rewritten to 0 by R if and only if p is an element of I.*

Theorem 7.3 *Suppose that the ordering among monomials is fixed, and let E and F be systems of Boolean equations. Then the reduced Boolean Gröbner bases of E and F are the same if and only if the generated ideals are the same.*

Note that the relation between the solutions and the ideal is complete for Boolean equations. Therefore, the reduced Boolean Gröbner bases satisfy the requirement in Section 5 perfectly.

8 Conclusion

The argument on semantics is mainly along the lines of that by Jaffar and Lassez [7]. Here we summarize the differences. We separated the constraint symbols from the predicate symbols. In general, a CAL programmer knows what function symbols and constraint symbols mean, but does not know how the system solves constraints. In this sense, these symbols are built-in in CAL. On the other hand, a programmer must know all about the predicate symbols because he introduces the symbols. Therefore, the semantics of constraint symbols and function symbols should be given a priori as a structure, while predicate symbols should be defined by a programmer. In this situation, separating the symbols at the beginning enables us to define the semantics naturally. In [7], the constraints are supposed to go ahead of the other literals in a clause. For flexibility, we did not assume this. We did not discuss finite definability, solution compactness, or satisfaction completeness, since we are not very interested in negation as failure, in particular, in CLP. There are many predicates which do not fit negation as failure. Even if a predicate fits such negation, there is most likely to be a decision procedure for the predicate, and in such a case, it seems to be more natural in CLP to incorporate the decision procedure into the constraint solver. Instead, we discussed the canonical forms of constraints, which are suitable as output from the system.

As shown in the second example, we can obtain an answer in the form of a relation among parameters, in particular, in the case where many parameters in a goal remain free. This effect is very similar to that of partial evaluation, e.g. [13], or the unfolding technique in logic programming, e.g. [14]. However, the result is more impressive and effective in CAL, since computation of Gröbner bases is much heavier and much more complicated than mere unification.

In the current version of CAL, the value of a variable in constraints may be (virtually) any algebraic number, i.e. a complex number which can be a solution of polynomial equations with integer coefficients. However, if a certain variable, say x, can take its value only in real numbers, then the constraint, $x^2 + 1 = 0$, is inconsistent. Therefore, if we have a powerful constraint solver which knows a lot about the smaller domain of real numbers, the execution time is expected to be reduced drastically for some practical problems. On the other hand, the user may want to write non-algebraic constraints, such as $\sin(x) = 1$, or $e^{\pi} = \pi$. In this case, it may be necessary to extend the domain to the set of all complex numbers.

Thus, there must be a tremendous variety of requirements in writing and solving constraints. To satisfy all unpredictable user requirements, the constraint solver should be designed to be completely open and customizable. According to this policy, the system is designed to accept the redefinition of a constraint solver suitable for the user's purpose. A user who remakes the constraint solver is required to clarify the language and the domain of his constraints according to Section 2 and to show that his constraint solver satisfies the criteria described in Section 5. At the very least, the user should implement an algorithm which determines the satisfiability of his constraints.

References

[1] Buchberger, B. (1983). Gröbner bases: An Algorithmic Method in Polynomial Ideal Theory. Technical Report, CAMP-LINZ.

[2] Colmerauer, A. (1987). Opening The Prolog III Universe: A New Generation of Prolog Promises Some Powerful Capabilities. *BYTE*, August, 177–182.

[3] Dincbas, M., Simonis H., and Van Hentenryck, P. (1987). Extending Equation Solving and Constraint Handling in Logic Programming. Technical Report TR-LP-2203, ECRC, February.

[4] Fikes, F. E. (1970). REF-ARF: A system for solving problems stated as Procedures. *Artificial Intelligence*, 1, 27–120.

[5] Heintze, N. C., Jaffar, J., Lim, C. S., Michaylov, S. Stuckey, P., Yap, R., and Yee, C. N. (1986). The CLP Progràmmer's Manual, Version 1.0. Department of Computer Science, Monash University.

[6] Hilbert, D. (1890). Über Die Theorie Der Algebraischen Formen. *Math. Ann.*, 36, 473–534.

[7] Jaffar, J., and Lassez, J-L. (1987). Constraint Logic Programming. Proc. 4th IEEE Symposium on Logic Programming.

[8] Jaffar, J., Lassez, J-L., and Maher, M. (1986). Logic Programming Language Scheme. In: (DeGroot, D., and Lindstrom, G. ed.) *Logic Programming: Functions, Relations and Equations*, Prentice-Hall.

[9] Jaffar, J., and Michaylov, S. (1985). Methodology and Implementation of a Constraint Logic Programming System. Technical Report TR 54, Department of Computer Science, Monash University, June.

[10] Lloyd, J. W. (1984). *Foundations of Logic Programming.* Springer-Verlag.

[11] Sato, Y., and Sakai, K. (1988). Boolean Gröbner Base. LA-Symposium in winter, RIMS, Kyoto University, February.

[12] Steele Jr., G. L., and Sussman, G. J. (1978). CONSTRAINTS. Technical Report 502, MIT AI Lab., Cambridge, Massachusetts.

[13] Takeuchi, A., and Furukawa, K. (1986). Partial evaluation of Prolog Programs and Its Application to Meta Programming. In:*Information processing 86, Dublin*, North-Holland.

[14] Tamaki, H., and Sato, T. (1984). Unfold/Fold transformation of Logic Programs. In:*Second International Logic Programming Conference, Uppsala.*

[15] van Emden, M. H., and Kowalski, R. A. (1976). The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23, (4), October.