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

Constraint Logic Programming Language CAL

Akira Aiba   Kô Sakai   Yosuke Sato   David J. Hawley
and Ryuzo Hasegawa

Institute for New Generation Computer Technology

4-28, Mita 1-Chome, Minatoku, Tokyo 108 JAPAN

October 6, 1988

Abstract

In this paper, we describe the current state of development of the constraint logic programming language CAL (*Contrainte Avec Logique*), and several future extensions. CAL supports the writing and solving as constraints of linear and non-linear algebraic polynomial equations, boolean equations, and linear inequalities. An implementation currently exists on the PSI machine. In the future, we are aiming to build a very flexible constraint logic programming language by combining multiple constraint solvers in one system.

1 Introduction

Constraint is one of the most important programming paradigms and is discussed in various field of knowledge processing from both the applicational and theoretical points of view. There are many advantages of constraint programming. The most outstanding feature of constraint programming is that it allows the declarative description of problems. That is, a problem is solved by indicating a goal without reference to the method by which it should be established.

In order to solve a problem by computers, we first have to precisely describe the problem. Therefore, in general, we have to determine a field and objects in it. In computer graphics, for example, the field is an Euclidean plane and the objects are points or other geometrical elements in the plane.

The set of objects in a certain field form a system. For example, the positions of points in Euclidean plane and their mutual distances satisfy a set of conditions. In general, these conditions are relations among the objects (or the parameters representing them).

Constraints are formulas representing such relations. The constraint programming paradigm is that conditions which must be satisfied by the objects in a program are described declaratively in terms of constraints [Ste-81], [Fik-70], [StS-78]. Constraints in a program are evaluated automatically and affect the execution of the program depending on their meaning.

When constraints are incorporated into logic programming, it is natural to call it constraint logic programming (CLP). Jaffar and Lassez were the first advocates of CLP [Jal-86], [JaL-87]. Similar paradigms (or languages) were proposed by Colmerauer [Col-82], [Col-87], and by Dincbas, Simonis, and van Hentenryck [Din-87]. Prolog programs are executed by a mechanism which includes unification as a major component. CLP is an attempt to increase the descriptive power of logic programming by employing constraint solving instead of unification in its execution mechanism. Unification itself is a kind of constraint solving. In this sense, CLP is a generalization of logic programming.

It is in the framework of logic programming that constraints give full play to their ability. In particular, the feature of declarative description, which is also a feature of logic programming, is preserved naturally and completely in CLP. There is other evidence that CLP is a natural extension of logic programming. For example, 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 control is described operationally. However, the operational semantics of CLP can be viewed as a simple generalization of the ordinary goal-reduction technique of logic programming.

Traditional logic programming possesses logical, functional, and operational semantics, which coincide with each other [EmK-76], [ApE-82], [Llo-84]. Jaffar and Lassez showed that CLP is a generalization of traditional logic programming in the sense that it possesses these three semantics [JaL-87]. In addition, they introduced algebraic semantics for CLP.

The execution steps of CLP programs depend upon the decision of whether or not a constraint is satisfiable in a given domain. However, we require more; the canonical forms of constraints should be computed if the constraints are satisfiable. The situation resembles that in ordinary logic programming, where unification decides the satisfiability of a set of equations in the Herbrand universe, and computes the most general unifier if the set is satisfiable. Since equations are typical constraints, the operational model of CLP is clearly an extension of that of logic programming. Moreover, the criteria of decidability of satisfiability, and the existence and computability of a canonical form, clarifies the requirements for constrain solvers in CLP and spurs on research on efficient methods to solve constraints in various domains.

This paper describes the theoretical foundation, implementation, and application of CAL (Contrainte avec Logique), which is the CLP language being developed at ICOT. Section 2 introduces constraint logic programming using CLP(R) as an example. Section 3 describes the semantics of CLP, focusing on the CAL system. Section 4 describes the current status of CAL: the system, language, constraint solver, and example programs. Section 5 discusses applications and future extensions of CAL.

2 Constraint Logic Programming Language

Constraint Logic Programming Languages originated with Prolog-II by Colmerauer [Col-82]. Subsequently, a scheme for constraint logic programming CLP(X) was proposed by Jaffar, Lassez, et. al. [JaL-86], and they gave its semantics in a logical frame-work. This scheme includes Prolog, Prolog-II, Prolog-III [Col-87], and CLP($\mathcal{R}$) [JaM-85] which was developed at Monash University in Australia as an example of the scheme. In a different approach, Dincbas proposed a constraint logic programming language in [Din-87], based on extended unification.

The difference between these constraint logic programming languages is the sort of constraints which they can handle. For instance, linear algebraic and boolean equations can be written in Prolog-III, while in CLP($\mathcal{R}$) algebraic equations and inequalities can be expressed, but only linear ones can be solved.

Constraint logic programming languages handle constraints by the following two methods: by extending the unification component of the logic programming language or by introducing a mechanism called the "Constraint Solver". We will discuss the processing of programs under the latter paradigm, which seems at least as general as the former.

Constraints are identified from other predicate-invocations by having a distinguished "constraint symbol", for example the "equality symbol" or "inequality symbol", as their principal functor. The execution model is similar to that for a logic programming language except that constraints are considered to be builtin predicates handled by the constraint solver.

To design a constraint logic programming language, the most important question is the selection of a computation domain and the selection of relations which will be handled as constraints. Dincbas proposed the following criteria to select a computation domain whose equations are embedded into unification [Din-87].

1. Each equation in the domain should be solved in a deterministic way without losing completeness.
2. Efficient equation solving methods should exist within the domain.
3. The computation domain should be used sufficiently often to justify its introduction inside unification.

We propose the following alternate set of criteria for the selection of computation domains over which equations and other constraints may be written [SaA-88]. The first three criteria correspond to those above.

1. Satisfiability (solvability) of constraints should be decidable.
2. An efficient algorithm to determine satisfiability should exist.
3. The computation domain should be used sufficiently often to justify supporting it directly in the language.
4. A canonical form for a constraint should exist if it is satisfiable, and the canonical form should be considered as the answer.

The last criterion has the following meaning. For example, the constraint `X+Y=1, X-Y=1` is equivalent to `X=1, Y=0`. In this case, if the latter can be considered the canonical form, the latter should be computed from the former. Canonical forms are described precisely in the next section.

We now consider an example. The following is a program to compute the product of two complex-numbers. This program is taken from the CLP($\mathcal{R}$) manual [Hei-86].

```
zmult(c(R1,I1),c(R2,I2),c(R3,I3)) :-
        R3 = R1*R2-I1*I2,
        I3 = R1*I2+R2*I1.
```

This clause means that a multiplication of a complex number `c(R1,I1)` and `c(R2,I2)` equals `c(R3,I3)`. Two equations in the body of the above clause are constraints, and the equality symbol indicates a constraint. A characteristic of the processing of this program is that the following three queries can be processed against the above clause.

```
?-zmult(c(1,1),c(1,2),c(R,I)).
?-zmult(c(1,1),c(R,I),c(-1,3)).
?-zmult(c(R,I),c(1,2),c(-1,3)).
```

For example, the processing of the second query proceeds as follows.

1. `?- zmult(c(1,1),c(R,I),c(-1,3)).` is unified against the clause head, and the substitution {R1/1, I1/1, R2/R, I2/I, R3/-1, I3/3} is obtained.

2. This substitution is applied to constraints (equations) in the body of the clause, and the following are obtained.

```
-1 = 1*R-1*I
3  = 1*I+R*1
```

When a new constraint is obtained, the constraint solver adds it to the set of constraints. More precisely, a newly obtained constraint is added to the previously computed constraint, and then computes the canonical form of the resulting constraint. Inconsistency of the new constraint with the previous constraint is treated in the same way as unification failure in a logic programming language.

3. The above system of equations is solved by Gaussian elimination, and the canonical form (solution) R=1, I=2 is obtained.

3 Semantics of CLP

3.1 CLP on Many Sorted Algebra and its declarative semantics

This section presents the basic notions needed to describe the semantics of CLP. The argument in this section is generally along the lines of that by Jaffar and Lassez [JaL-87], but is different in the details.

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 a *signature*, is assigned to each function, predicate, and constraint symbol. We write $v : s$, $f : s_1 s_2 \dots s_n \rightarrow s$, and $p : s_1 s_2 \dots s_n$ if a variable, v, has a sort, s, if a function symbol, f, has a signature, $s_1 s_2 \dots s_n$, and a sort, s, and if a predicate or constraint symbol, p, has a signature, $s_1 s_2 \dots s_n$, 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 $f : s_1 s_2 \dots s_n \rightarrow s$, and $t_1, t_2, \dots, t_n$ are terms of sorts $s_1, s_2, \dots, s_n$ respectively, then $f(t_1, t_2, \dots, t_n)$ is a term of sort s.

Atomic formulae and *atomic constraints* are defined as follows.

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

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

We write $t : s$ if a term t has a sort s. The sets 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.

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

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 be a function over $T(F, V)$ and $A(C, F, V)$. Then $t\Theta \in D(s)$ if t is a term of sort s, and $p\Theta$ is **false** or **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 be 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 $p : -p_1, p_2, \ldots, p_n$ $(n \geq 0)$, where p is an atomic formula and each p_i is either 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 $(p : -p_1, p_2, \ldots, p_n) \in L$, and for any assignment, Θ, $p_1\Theta I = p_2\Theta I = \cdots = p_n\Theta I = \mathbf{true}$ implies $p\Theta I = \mathbf{true}$.

3.2 Functional Interpretation of a Program

First, we extend the function given by van Emden and Kowalski [EmK-76] for CLP. Let there be a program, L. Based on an interpretation, I, we can define another interpretation, J, as follows.

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

Since interpretation J is dependent on program L and interpretation I, we 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 $p : s_1 s_2 \ldots s_n$, and for every element $d_1 \in D(s_1), d_2 \in D(s_2), \ldots, d_n \in D(s_n)$, if $I(p)(d_1, d_2, \ldots, d_n) = \mathbf{true}$, then $J(p)(d_1, d_2, \ldots, d_n) = \mathbf{true}$. Proof of the following proposition is 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.

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

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, $d_1 \in D(s_1), d_2 \in D(s_2), \ldots, d_n \in D(s_n)$, $T \uparrow 0(p)(d_1, d_2, \ldots, d_n) = \mathbf{false}$. 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, $d_1 \in D(s_1), d_2 \in D(s_2), \ldots, d_n \in D(s_n)$, $T \downarrow 0(p)(d_1, d_2, \ldots, d_n) = \mathbf{true}$.

It is easy to show the following.

$$T \uparrow 0 \leq T \uparrow 1 \leq T \uparrow 2 \leq \ldots$$
$$T \downarrow 0 \geq T \downarrow 1 \geq T \downarrow 2 \geq \ldots$$

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 on 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.

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

In particular, $Q \uparrow 0 = \emptyset$ and $Q \downarrow 0 = \mathbf{QA}$. The following are also routine.

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

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

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

```
|S|(p) (d_1, d_2, ..., d_n)
      if there is a QA-pair (p(t_1, t_2, ..., t_n) : -C) ⊆ S
         and an assignment, ΘI,
         such that d_1 = t_1Θ, d_2 = t_2Θ, ..., d_n = t_nΘ
         and Θ is a solution of C
      then true
      else  false
```

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, $\mathrm{lfp}(T, L) = |\mathrm{lfp}(Q, L)|$ and $\mathrm{gfp}(T, L) = |\mathrm{gfp}(Q, L)|$.

3.3 Operational Interpretation of a Program

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. When $n = 0$, the goal comprising only a satisfiable constraint is called a *successful* goal. 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, $SS(L)$.

$$\begin{aligned} SS(L) = \{\ (p : -C) \in \mathbf{QA} \mid \ & \text{there exists an SLD-resolution sequence which} \\ & \text{begins with the goal, } p; \emptyset, \\ & \text{and ends with the successful goal, } C\}. \end{aligned}$$

Theorem 3.1 *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.

3.4 Constraint Solving and Canonical Forms

According to the operational model of CLP described in the previous section, decidability 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 a satisfactory form of output from the system. 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. However, the answer that users actually want in many cases is something like $\{x = 2,\ y = 1\}$. In this sense, *constraint solving* should not be a mere decision on the satisfiability of constraints, but a 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 to compute the canonical form of the union, D, of constraints instead of merely making the union. Actually, the unification procedure 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.

3.5 Examples of Language and Domain

A typical domain of CLP is the field of all the algebraic numbers, of which the formal language, for example, is defined as follows.

$$\begin{aligned}
S &= \{\mathbf{A}\}\\
F &= \{\times : \mathbf{AA} \to \mathbf{A}, + : \mathbf{AA} \to \mathbf{A}\} \cup \{\text{fraction} :\to \mathbf{A}\}\\
C &= \{=\}\\
P &= \{\text{string starting with a lowercase letter}\}\\
V &= \{\text{string starting with an uppercase letter}\}
\end{aligned}$$

We assume that there is only one sort $\mathbf{A}$ of algebraic numbers for simplicity. We define a structure for the above language as follows.

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

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

The next example is CLP in a Boolean algebra (or more precisely, in a Boolean ring). The language and the structure is defined, for example, as follows.

$$\begin{aligned}
S &= \{\mathbf{B}\}\\
F &= \{\wedge : \mathbf{BB} \to \mathbf{B}, \oplus : \mathbf{BB} \to \mathbf{B}, \bot :\to \mathbf{B}, \top :\to \mathbf{B}\}\\
C &= \{=\}\\
P &= \{\text{string starting with a lowercase letter}\}\\
V &= \{\text{string starting with an uppercase letter}\}
\end{aligned}$$

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

Ordinary Prolog can be defined as an CLP language as follows.

$$\begin{aligned}
S &= \{\mathbf{H}\}\\
F &= \{\text{string starting with a lowercase letter}\}\\
C &= \{=\}\\
P &= \{\text{string starting with a lowercase letter}\}\\
V &= \{\text{string starting with an uppercase letter}\}
\end{aligned}$$

$$\begin{aligned}
D(\mathbf{H}) &= \text{ Herbrand universe } = \{\text{ground term}\}\\
D(F) &= \text{ syntactic construction}
\end{aligned}$$

4 CAL

4.1 The current status of CAL interpreter

The present CAL is not a single language, but a family of languages over different computation domains. Languages within this family are as follows.

On DEC2060, there are three CAL interpreters: Algebraic CAL for algebraic equations, Boolean CAL for boolean equations, and Linear CAL for linear algebraic equations and inequalities. On the PSI, there are: Algebraic CAL, Boolean CAL, and Typed CAL; the last supports constraints in many sorted algebra described in Section 3.

In the actual CAL interpreter, some semantically impure features have been added for the convenience of users, as was done for Prolog. Since we will be discussing the actual CAL interpreter, we will also mention these features where it seems appropriate.

On both DEC 2060 and PSI, each CAL interpreter comprises the following components.

1. Pre-processor

   Translates CAL source program and query into DEC-10 Prolog on DEC 2060, and into ESP on PSI.

2. Constraint solver

   Receives constraints, and computes their canonical form.

4.2 Evaluation of CAL programs

The current CAL implementations are layered on top of Prolog systems using preprocessors. The most significant difference between logic programming languages and constraint logic programming languages is that the former maintains the solution of unifications as bindings, and the latter maintains the canonical form of a constraint (a set of atomic constraints). Both language classes, however, must maintain these partial solutions during both forward (resolution) and backward (backtracking) execution of their programs. The preprocessing of a CAL program is done in similar way to that for a DCG (*Definite Clause Grammar*); namely, a pair of variables is added as the last arguments of each clause, for the input of the old, and the output of the new constraint.

A CAL program

```
zmult(c(R1,I1),c(R2,I2),c(R3,I3)) :-
        R3 = R1*R2-I1*I2,
        I3 = R1*I2+R2*I1.
```

is translated into the following Prolog program.

```
zmult(c(R1,I1),c(R2,I2),c(R3,I3),V0,V2) :-
        constraint(R3 = R1*R2-I1*I2,V0,V1)
        constraint(I3 = R1*I2+R2*I1,V1,V2).
```

The meaning of predicate `constraint/3` is to add a newly obtained constraint, which is passed through its first parameter to the canonical constraint, which is passed through the second parameter, to compute a new canonical constraint, and to unify it to the third parameter.

In the same style, the preprocessor adds two extra variables to all predicates except those built into Prolog. Since the output of the translation is a Prolog program, we only need to provide a predicate `constraint` satisfying the above specification.

4.3 Algorithms for Constraint Solving

As shown in the example in section 2, algorithms for constraint solving should have the ability to solve constraints incrementally. Since constraints are obtained one by one, canonical forms are obtained by transforming former canonical forms and newly obtained constraints.

We now describe the algorithms for the constraint solving component of the following CAL interpreters.

1. Algebraic CAL
2. Boolean CAL
3. Linear CAL

4.3.1 Algebraic CAL

In this section, we describe the algorithm for Algebraic CAL. The Buchberger algorithm for computing Gröbner bases of polynomials, which has been used in recent years in computer algebra and geometrical theorem proving, is utilized as the constraint solving algorithm.

Buchberger introduced the concept of Gröbner bases, and presented the algorithm to compute the bases of input polynomials [Buc-83].

Without loss of generality, we can assume the form of equations to be $p = 0$. Let $E=\{p_1 = 0, p_2 = 0, \ldots, p_n = 0\}$ be a system of polynomial equations, and let I be the ideal generated from $\{p_1, p_2, \ldots, p_n\}$ in the polynomial ring. The following theorem shows the relationship between elements of I and solutions of E.

Theorem 4.1 (Hilbert's zero point theorem)
Let p be a polynomial. Every solution of E is also a solution of $p = 0$, if and only if p^n is an element of I for some integer n [Hil-90].

The next corollary is important for determining the solvability of constraints.

Corollary 4.1
E has no solutions if and only if $1 \in I$.

By the above considerations, the problem of satisfying the polynomial equation $p = 0$ under constraints E can be transformed into the problem of the determination of whether a polynomial p^n belongs to the ideal generated from E or not. Buchberger gave an algorithm to determine whether a polynomial belongs to the ideal or not.

In the system of polynomial equations, each equation can be considered as a rewriting rule which rewrites the maximal monomial to a remaining polynomial under a certain ordering of monomials. When the left-hand sides of an arbitrary pair of rewrite rules are not mutually prime, their least common multiplier can be rewritten to two polynomials by two rules. The pair of these polynomials is called the *critical pair* of these two rules. Among critical pairs, there may be ones whose rewriting are not confluent. This kind of critical pair is called *divergent*.

Let E be a set of given equations, and R be a set of rewriting rules. A Göbner base of E is computed as the final R by the following algorithm:

1. $R \leftarrow \emptyset$
2. For each equation $l - r$ in E, simplify it by rewriting rules in R and arithmetic operations. Let e be an equation resulting from this simplification. If $e \equiv 0$, then this equation is thrown away. Otherwise, replace the original equation $l = r$ in E by $e = 0$.
3. If $E = \emptyset$, then end.

4. Select an equation $e = 0$ in E.

5. Let l' be the maximal monomial in e under a certain monomial ordering. Solve $e = 0$ with respect to l'. Let $l' = r'$ be the result.

6. Add a rule $l' \to r'$ to R.

7. Add every divergent critical pair of rules in R to E as equations.

8. Go to 2.

The next theorem states the relationship between ideals and Gröbner-bases.

Theorem 4.2 (Buchberger)
Let R be a Gröbner-base of system of equations, $\{p_1 = 0, p_2 = 0, \ldots, p_n = 0\}$, I be an ideal generated from $\{p_1, p_2, \ldots, p_n\}$. Then a polynomial p belongs to I if and only if p can be rewritten into 0 by R.

The next theorem certify the validity of Gröbner-base as canonical form of constraints. Here, irreducible Gröbner-base is the base in which no couple of rules can rewrite each other.

Theorem 4.3
Suppose that the monomial ordering is fixed. Let E and F be systems of equations. If an ideal generated from E is same as that from F, then the irreducible Gröbner-base of E is same as that of F.

In Algebraic CAL, constraints can have "=" and "==" as their constraint symbols. A constraint $f = g$ indicates a so-called active constraint, and the constraint solver computes the new canonical forms by adding this constraint. That is to say, the Gröbner-base is modified to satisfy $f = g$. On the other hand, $f == g$ is a kind of a passive constraint, and the constraint solver checks whether $f = g$ is satisfied or not under the corrected constraints. "Passive" as used above has a different meaning from that used by Dincbas. The meaning here is constraints without modifying Gröbner-base. This is analogous to "==" in Prolog, which does not cause unification. Note that the relationship between constraints and ideal is not complete. For instance, $X = 0$ is satisfied under constraints $\{X^2 = 0\}$, but it does not belong to the ideal generated from $\{X^2 = 0\}$. Therefore, check of the latter constraints needs more careful use of the Gröbner-base than rewriting with it.

This "==" is added to the system for the convenience of users.

4.3.2 Boolean CAL

In this section, we describe the algorithm used in the constraint solver for Boolean CAL. In Boolean CAL, boolean equations can be written, and can be processed as constraints. The typical computation domain is the set of truth-values.

There are many decision procedures for the solvability of boolean equations. Among them, the typical one is *semantic unification* employed by Dincbas [Din-87]. We, in CAL, use the approach of Boolean Gröbner-bases [SaS-87] which is obtained by slight modification of Buchberger algorithm. We think that this method has advantages in the following points:

1. This algorithm computes answer constraints without introducing extra variables. Thus answer constraints can be understood easily.

2. There exist canonical forms for constraints, and their meaning is clear.

In Boolean CAL, given constraints are translated into boolean polynomials, and then the Gröbner-base of them are computed. This is the evaluation of constraints in Boolean CAL.

A boolean equation is a polynomial whose coefficients are 0 or 1. Moreover, we can define it so that degree of each variable in every monomial is 1. That is to say, there are no monomials such as x^3y in boolean equations. Instead, it must be written as xy. This is caused by the fact that every elements in boolean rings are idempotent.

The most significant difference between boolean equations and ordinary algebraic equations is that the constraint evaluation in the former can be completely described by the ideal.

Corresponding to the theorem 4.1, the next theorem holds.

Theorem 4.4
Let p be a boolean polynomial. Every solution of the system of boolean equations $E = \{p_1 = 0, \ldots p_n = 0\}$ is also a solution of $p=0$ if and only if p is an element of I[1].

The algorithm to compute boolean Gröbner-bases is almost the same as that for polynomial rings, excepting the following points.

For a boolean polynomial $AX + Z$, $AZ + Z$ is called its *self critical pair*, where A, X, and Z are a variable, a monomial, and a polynomial, respectively, and AX is the maximal monomial in $AX + Z$. In boolean ring, $x + x = 0$ and $xx = x$,

$$AX + Z = 0 \Rightarrow (AX + Z)(A + 1) = AZ + Z = 0,$$

which certify that $AZ + Z$ should be a critical pair.

[1]This property holds in arbitrary boolean rings

By using the concept of the self critical pair, the algorithm to compute boolean Gröbner-bases can be obtained by modifying that for Gröbner-bases as follows.

In 7;, not only divergent critical pairs but also divergent self critical pairs are added to E as equations.

For boolean Gröbner-bases obtained as the above, Theorem 4.2 and 4.3 are hold as they are. Refer to [SaS-88] for preciseness.

4.3.3 Linear CAL

In this section, we describe constraint solving in Linear CAL. In Linear CAL, linear equations and linear inequalities can be written and processed as constraints. A number of constraint solvers based on either the simplex or Min-Max method have ben proposed for their solution. We are now developing a simplex-method based constraint solver which satisfies our criteria for computation domains.

However, since the canonical form at present is not easily understandable by users, we have to consider the following points:

1. The method of output of answer constraints.
2. Canonical forms of constraints.
3. Constraint solving algorithm.

4.4 Programming Examples

In this section, we describe the description power of CAL by showing programming examples for each of its constraint solvers.

4.4.1 Programming Examples in Algebraic CAL

As we described above, the major characteristic of Algebraic CAL is the ability to solve non-linear equations. The following examples include one on the edges and surfaces of triangles, and several of geometrical theorem proving.

Example 4.1 (Geometrical Theorem Proving)
The following theorem is considered.

Theorem 4.5
Let ABCD be an arbitrary quadrangle, and E, F, G, and H be mid-points of edges AB, BC, CD, and DA, respectively. Then the quadrangle EFGH is a parallelogram.

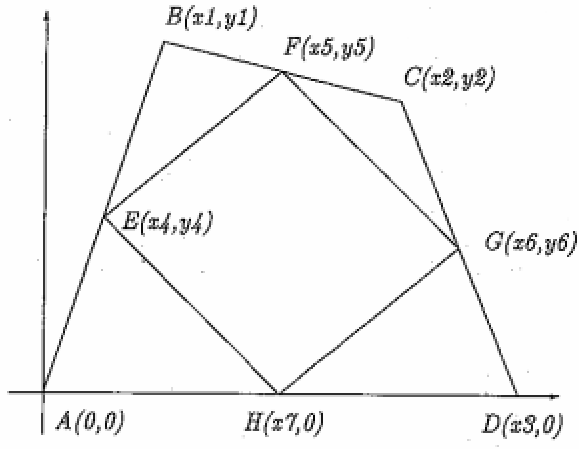

Fig.-1: Geometrical Theorem Proving

To prove the theorem, we transform this geometrical problem to an algebraic problem by introducing Cartesian coordinate system. A mid-point (x_2, y_2) of a segment $(x_1, y_1) - (x_3, y_3)$ can be represented by equations $x_2 = (x_1 + x_3)/2, y_2 = (y_1 + y_3)/2$. The fact that segment $(x_1, y_1) - (x_2, y_2)$ is parallel with segment $(x_3, y_3) - (x_4, y_4)$ can be represented by an equation $(y_2 - y_1)/(x_2 - x_1) = (y_4 - y_3)/(x_4 - x_3)$.

These equations are represented by CAL as follows:

```
mid(X1,Y1,X2,Y2,X3,Y3) :-
        2*X2 = X1+X3,
        2*Y2 = Y1+Y3.
para(X1,Y1,X2,Y2,X3,Y3,X4,Y4) :-
        (X1-X2)*(Y3-Y4)==(Y1-Y2)*(X3-X4).
```

By evaluating the following query against the above program, the given theorem is proven.

```
?-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).
```

The basic idea of this program and query is to certify that the two pairs of segments, whose endpoints are constrained to be the midpoints of the original quadrilateral, are parallel.

As mentioned before, we introduce the constraint symbol "==" in the above program for programming convenience.

Example 4.2 (Heron's formula)
The problem is to obtain the relationship between the length of three edges of a triangle and its surface. For an arbitrary triangle, let l be the length of its base edge,

h be its height, and s be its surface. Then the following relation holds: $l * h = 2 * s$, implemented as the predicate `sur`.

Then, we describe the Pythagorean Theorem by the predicate `right`. *Let a and b be lengths of edges which are connected to the square corner, and c be the length of the other edge. If $a^2 + b^2 = c^2$, then that triangle is a right triangle. In the following program, n^m means n^m.*

Moreover, the fact that an arbitrary triangle can be divided into two right triangles is described by the predicate `tri` *(see Fig-2).*

```
sur(L,H,S) :- L*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(C,H,S).
```

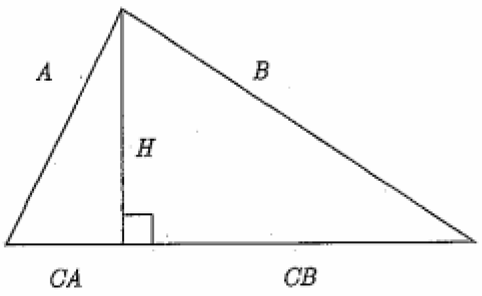

Fig-2: Three edges of a triangle and its surface

When a goal in which all arguments are free `tri(a,b,c,s)` is given, this program outputs the Gröbner-base with 7 rules. Among them, there is the following rule which contains only a, b, c, and s. This is equivalent to the Heron's formula.

```
c^2 = (-c^4+ -1*a^4+2*(2*b^2*a^2)+ -1*b^4
      +2*(2*c^2*a^2)+2*(2*c^2*b^2))/16
```

This program will also run with instantiated queries. For instance, if a goal `tri(3,4,5,s)` is given, then `s^2=36` is output.

Example 4.3 (Conditional Extremum)
Compute the conditional extremum using Lagrange's method of indeterminate coefficients.

The following CAL program realizes Lagrange's method of indeterminate coefficients.

```
ex(F, Constraint,Vars) :-
                 lag(Constraint, Lag),
                 difs(Vars, F, Lag).
lag([ ], 0) :- !.
lag([L=R |Cs], Mult*(L-R)+Lag) :-
                 L=R,
                 lag(Cs, Lag), !.
difs([ ], _, _) :- !.
difs([Var |Vars], F, Lag) :-
                 dif(F, Var)=dif(Lag, Var), !,
                 difs(Vars, F, Lag).
```

The first argument for a predicate `ex` is an objective function whose extremum will be computed, the second argument is a list of conditions on the computation, and the third argument is a list of symbols whose values can be modified (that is to say, this is a list of variables). `dif(F,Var)` denotes a polynomial obtained by differentiating a polynomial `F` by a variable `Var`. This notation is built into the CAL interpreter for the convenience of users.

Strictly speaking, `dif(F,Var)` is not a polynomial, but a term in the Prolog sense, and so it is misleading to describe it as a constraint. However, we introduce it for programming convenience.

This program can be used to solve the following problem.

Problem 4.1
Divide a circle into two fans by two radial cuts, making two cones. The problem is to obtain the angle between the two radial cuts which maximizes the sum of the volumes of the two cones.

We can assume a circle of radius 1, since the answer doesn't depend on the size of the circle. After making the first cut, make the second one at a distance $\pi + r$ along the circumference, measured in one direction, $\pi - r$ in the other. Suppose the cones have height sA and sB respectively. Then, factoring out constants, we obtain the following query.

```
ex((1/2+r)^2*sA+(1/2-r)^2*sB,
   [sA^2+(1/2+r)^2 = 1, sB^2+(1/2-r)^2 = 1],
   [sA, sB]).
```

This program outputs a Gröbner-base of three rules, among them the following degree-7 polynomial which contains r as its only variable. However, if both sides of this equation are divided by r, then it becomes a cubic over r^2, whose roots can be obtained easily.

```
r^7 = (29/12)*r^5+(-17/48)*r^3+(5/576)*r
```

4.4.2 Programming Example of Boolean CAL

As we mentioned above, Boolean CAL handles boolean equations. Here we present the verification of a logic circuit taken from [Din-87] as an example of its use.

Example 4.4 (Cross Circuit)
The problem is to prove that the following circuit is a cross circuit.

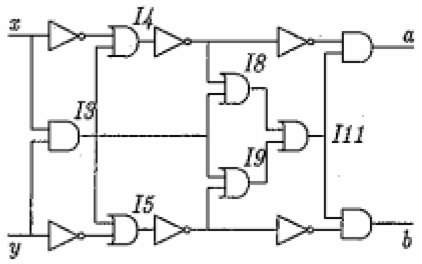

Fig.-3: Cross Circuit

To prove that the circuit is a cross circuit, we must do the following. First of all, we describe the specification of the circuit in terms of boolean equations. Secondly, the relation between input-terminals and output-terminals are described. Accordingly, the following program is obtained.

```
cir(X,Y,A,B) :-
      I4 = ~XVI3, I3 = X∧Y, I5 = ~YVI3,
      I8 = ~I4VI3, I9 = ~I5∧I3,
      A = I4∧I11, I11 = I8VI9, B = I5∧I11.
```

The following query is evaluated against the above program. In the query, all arguments are left free.

```
?- cir(x,y,a,b).
```

The resulting output proves the result.

```
x=b

y=a
```

5 Extensions of CAL

At present, we have three constraint solvers for CAL. Moreover, we implement Typed CAL in which users can use constraints on several types of objects simultaneously. In this section, we describe Typed CAL, and some future extensions.

5.1 Typed CAL

The basic idea of Typed CAL is to realize constraints in many sorted algebras discussed in Section 3, and thus allow users to use multiple constraint solvers simultaneously. Typing is introduced to indicate the sorts of parameters. In execution of a program, a suitable solver is selected automatically according to the type of each atomic constraint. At the time that we designed Typed CAL, the Buchberger Algorithm to compute Gröbner bases, and the algorithm to compute boolean Gröbner bases were available. We made an experimental implementation of Typed CAL on the PSI–machine with these two constraint solvers. Recently, a third constraint solver based on linear programming has been implemented, and there are plans to implement another constraint solver for real closed fields. We intend to add two new types corresponding to these new solvers to Typed CAL.

The indication of the type for each constraint will be made as follows. Available types will be:

1. `alg` (algebraic number – To invoke a constraint solver for Gröbner-bases),
2. `bool` (To invoke a constraint solver for boolean Gröbner-bases),
3. `lin` (To invoke a constraint solver for linear equations and linear, and inequalities),
4. `real` (To invoke a constraint solver for the real closed field),
5. `_` (To invoke ordinary unification for ordinary Prolog term)

The type of a constraint is indicated by placing a "`:`" followed by the appropriate type-name after the constraint.n

To allow users to write formulae in the head of a clause, we will introduce type declarations for predicates. For instance, type declaration `:- type p(alg, bool, _).` means that the first argument of `p` is an algebraic constraint to compute Gröbner-bases, the second argument is a boolean constraint, and the third one is an ordinary argument in Prolog. For instance, we may write the following program:

```
p(X+Y, ~ B, C) :- Q .
```

Suppose that we evaluate the following query against the program.

```
p(5, true, a).
```

Unification between the query and clause-head succeeds, and the algebraic constraint `X+Y=5`, the boolean constraint `~ B = true`, and the substitution `{C/a}` are obtained. This result is brought about by the preprocessing of the above program into the following clause.

```
p(D1, D2, C) :- X+Y=D1:alg, ~ B = D2:bool, Q
```

Thus, the above constraints and substitution are obtained.

By introducing types into predicates, we can implement very similar features to those provided by Dincbas' semantic unification [Din-87].

The following is an example of using type declarations. The next clause solves the *man and horse* problem, using constraint typing, but without predicate type declarations.

```
mah(Man,Horse, Legs, Heads) :-
        Heads = Man+Horse,
        Legs = Man*2+Horse*4.
```

By introducing a predicate type declaration, the above program can be written in the form of a unit clause as follows.

```
:- type mah(_, _, alg, alg).
mah(Man, Horse, Man*2+Horse*4, Man+Horse).
```

5.2 The requirement for CAL from geometry theorem proving

In this section, we discuss geometrical theorem proving, a typical application of CAL. Elementary geometry can be classified into the following hierarchy of classes, arranged in order of increasing size. i) affine geometry, ii) pre-Euclidean geometry, and iii) Euclidean geometry. Example 4.1 belongs to i). In general many theorems in class(i) can be proved by using a simple Gröbner base method. The case of class(ii) is more complicated.

Example 5.1 *Three perpendiculars drawn from three vertices of a triangle converge. Let A,B,C be three vertices of a triangle, D,E be the feet of perpendiculars drawn from A,B respectively, and F be the intersection point of the lines AD and BE.*

We put A=(u₁, 0), B=(u₂, 0), C=(0, u₃), D=(x₁, x₂), E=(x₃, x₄), F=(x₅, x₆). Then the conditions are expressed as follows.

$$\begin{array}{ll}
D \text{ is on } CB & h_1 = u_2x_2 + u_3x_1 = 0 \\
AD \perp CB & h_2 = u_3x_2 - u_2x_1 + u_1u_2 = 0 \\
E \text{ is on } CA & h_3 = u_1x_4 + u_3x_3 = 0 \\
BE \perp CA & h_4 = x_4u_3 - u_3u_1) + u_1u_2 \\
F \text{ is on } AC & h_5 = x_1x_6 - x_2x_5 - u_1x_6 + u_1x_2 = 0 \\
F \text{ is on } BE & h_6 = x_3x_6 - x_4x_5 + u_2x_4 - u_2x_6 = 0
\end{array}$$

Convergence of the three perpendiculars is expressed by the next equation.

$$g = x_5 = 0$$

If we evaluate $g = 0$ under the constraint $h_1 = 0, \dots h_6 = 0$, by a CAL program in the same manner as example 4.1, the system replies "NO". The problem is that when

$$u_1 = u_2 = u_3 = 0,$$

the constraint becomes $x_1x_6 - x_2x_5 = 0, x_3x_6 - x_4x_5$. Clearly $g = 0$ does not obtain under this degenerate condition, where A, B, C are congruent.

For solving this kind of nondegenerate problem, the well-known Ritt's Decomposition Algorithm [Rit-38] is very effective. Not only can we make a complete prover of class(ii) problems using this algorithm, but also it is very powerful for decomposing algebraic constraints. The reason we can make a complete prover of class(ii) problems fairly easily is that it has models over the complex numbers. On the other hand class(iii) has models over the reals.

We are now implementing this algorithm as a constraint solver for CAL.

Example 5.2
Take two squares ACDE, BCFG *outside of a given triangle* ABC. *Let* M *be the midpoint of* AB. *Then* DF=2CM.

In order to express "outside of" algebraically, we need inequalities over real numbers.

It is well known that we can have a complete prover for class(iii) using Tarski's algorithm or Collins' quantifier elimination algorithm [Col-75]. However these algorithms are not impractical due to their time and memory inefficiency. Implementing efficient algorithms dealing with the real closed field to meet the needs of geometry theorem proving is our future task.

5.3 parallelization of CAL

Parallelization of CAL is still in the stage of preliminary investigation, and we do not have a definite policy under which all researches are conducted. Here, we discuss problems in parallel constraint logic programming based on the current CAL.

Programs in a constraint logic programming language fit into one of the following two cases:

1. There is a definite constraint set (deterministic type). The purpose of computation is to obtain a unique solution. All the examples in this paper are of this type.

2. There are several possible constraint sets (nondeterministic type). Among these, a searching is done for a consistent constraint set. This is a kind of search problem, and is seen in applications like CAD and some kinds of puzzles.

Problems of the former type do not need exhaustive search and, therefore, fit committed-choice type languages like GHC. Those of the latter type definitely depend on the search mechanism povided with logic programming languages, and need "OR-parallel" execution or an equivalent. If an exhaustive search type problem is programmed in GHC, for example, conversion to a committed choice type program or simulation of exhaustive search by an interpreter is necessary. We aim at research into (i) parallelization of Constraint Solver, (ii) parallelization of Inference Engine, and (iii) design of a parallel CLP language. Before researching of (ii) and (iii), however, the characteristic of application programs should be investigated.

If the future parallel CLP language is implemented in GHC, it may be reasonable to change the operational semantic of CLP to fit committed choice execution, though practicality of this approach is highly dependent on application characteristics. If this is the case, the specifications for guards and suspension rules should be similar to those for GHC. Our current opinion is as follows. In a GHC-like CLP language, we can write passive constraints in the guard part of a clause and active constraints in the body part. Committed choice of a clause is caused by satisfaction of all the head unification and all the passive constraints in the guard parts. When they are not satisfied currently but are satisfiable, the execution of the clause is suspended until a new active constraint is obtained. A lot of GHC features are reserved by this CLP version of GHC, and so this approach makes it easy to implement a system for constraint programs of deterministic type.

6 Conclusion

CAL is still under development, and its final shape is not yet clear. However, we can see three future directions for development.

1. implementation new constraint solvers

   For example, a constraint solver of linear inequalities, or specializations of the quantifier elimination algorithm mentioned in section 5.

2. combining constraint solvers for application

   Among the applications of Gröbner bases, the automated theorem prover for elementary geometry is one of the most successful topics of research. We are planning to develop a theorem prover for geometry as a CAL application. Since constraint solvers based on Gröbner bases are not strong enough to support a prover, we have to develop subalgorithms of the quantifier elimination. Furthermore these must be available in a uniform environment, which should be realized by Typed CAL. In this uniform environment, it would be better to make hierarchies of constraint solvers, since a general algorithm is usually less efficient than its specializations.

3. Parallel CAL

   The final form of CAL will be a parallel constraint programming language with full functions obtained as the results of the researches described in 1 and 2. In the design of parallel CAL, however, deep consideration of cost-effectiveness is necessary. In particular, the characteristics of algorithms used in application programs should be carefully investigated. For the moment, a few prototype languages will be designed and implemented experimentally according to characteristics, for instance deterministic or non-deterministic, of application fields, through which we can gain our experience in order to unify those prototypes into parallel CAL.

The argument on semantics of CAL [SaS-88] is mainly along the lines of that by Jaffar and Lassez [JaL-87]. 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 to 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 the programmer. In this situation, separating the symbols at the beginning enables us to define the semantics naturally. In [JaL-87], 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 constraint logic programming. There are many predicates for which negation as failure is inappropriate. 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 constraint logic programming to incorporate the decision procedure into the constraint solver. Instead of these three topics, we discussed the canonical forms of constraints as suitable output from a constraint logic programming system.

Both CAL and CLP(R) 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. [TaF-86], or the *unfolding technique* in logic programming, e.g. [TaS-84]. 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 present version of Algebraic CAL, the imaginary value of each variable in a constraint is an algebraic number i.e. a complex number which could be a solution of a polynomial equation with integers as its coefficients. If we have a constraint solver over a more restricted domain, the efficiency of solving some problems is drastically improved. For example, when we know a variable x can take only a real number as its value, a constraint like $x^2 + 1 = 0$ leads to a contradiction. On the other hand, we might want to have non-algebraic constraints like $\sin(x) = 1$ or $e^x = \pi$. We might have to extend the domain to all complex numbers. There are many requirements other than these for describing and solving constraints. In order to satisfy these requirements, constraint solvers must be completely changeable by users. We designed the system so that users can define constraint solvers for their own purpose, by making the semantics of the domains precise, and implementing corresponding solvers.

[References]

[ApE-82] K. R. Apt, and M. H. van Emden, "Contributions to the Theory of Logic Programming", *JACM*, 29(3), July, 1982, pp.841-862.

[Buc-83] B. Buchberger, "Gröbner Bases: an Algebraic method in Polynomial Ideal Theory", Technical Report, CAMP-LINZ, 1983.

[Col-75] G. E. Collins, "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition," Lecture Notes In Computer Science 33, 1975.

[Col-82] A. Colmerauer, "PROLOG-II – Reference Manual and Theoretical Model", Internal Report, Groupe Intelligence Artificille, Universite Aix-Marseille II, October, 1982.

[Col-87] A. Colmerauer, "Introduction to Prolog-III", ESPRIT'87, Achievements and Impact, Proc. of the 4th Annual ESPRIT Conference, Brussels, September 28-29, 1987, North-Holland.

[Din-87] M. Dincbas, H. Simonis, and P. van Hentenryck, "Extending Equation Solving and Constraint Handling in Logic Programming", ECRC Internal Report, IR-LP-2203, February, 1987.

[EmK-76] M. H. van Emden, and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language", *JACM*, 23(4), October, 1976, pp.733-742.

[Fik-70] R. E. Fikes, "REF-ARF: A System for Solving Problems stated as Procedures", *Artificial Intelligence*, 1, 1970, 27-120.

[Hei-86] N. Heintze, J. Jaffar, C. S. Lim, S. Michaylov, P. Stuckey R. Yap, and C. N. Yee, "The CLP Programmer's Manual – Version 1.0", Department of Computer Science, Monash University, 1986.

[Hil-90] D. Hilbert, "Über die Theorie der algebraischen Formen", *Math. Ann.* 36, pp.473-534, 1890.

[JaL-86] J. Jaffar, and J-L. Lassez, "Constraint Logic Programming", IBM Thomas J. Watson Research Center, Internal Memo, 1986.

[JaL-87] J. Jaffar, and J-L. Lassez, "Constraint Logic Programming",*4th IEEE Symposium on Logic Programming*, 1987.

[JaM-85] J. Jaffar, and S. Michaylov, "Methodology and Implementation of a Constraint Logic Programming System", TR 54, Department of Computer Science, Monash University, June, 1985.

[Llo-84] J. W. Lloyd, "Foundations of Logic Programming", Springer-Verlag, 1984.

[Rit-38] R. F. Ritt, "Differential Equation from Algebraic Standpoint," AMS Colloquium Publications Volume 14, New York, 1938.

[SaA-88] K. Sakai, and A. Aiba, "CAL : A Theoretical Background of Constraint Logic Programming and Its Applications (In Japanese)", Information Processing Society of Japan, vol. 88, No. 8, 88-SF-24, pp. 9–17、Feb. 12, 1988.

[SaS-88] Y. Sato, and K. Sakai, "Boolean Gröbner Bases", LA-Symposium, February, 1988.

[Ste-81] M. Stefik, "Planning with Constraints", *Artificial Intelligence*, 16, 2, 1981.

[StS-78] G. L. Steele, and G. J. Sussman, "Constraints", MIT AI Lab Memo 502, Cambridge, Massachusetts, 1978.

[TaF-86] A. Takeuchi, and K. Furukawa, "Partial evaluation of Prolog Programs and Its Application to Meta Programming", *Information processing 86*, Dublin, North-Holland, pp. 415–420, 1986.

[TaS-84] H. Tamaki, and T. Sato, "Unfold/Fold transformation of Logic Programs", *Second International Logic Programming Conference*, Uppsala, 1984.