Chapter 1

RISC-CLP(Real): Logic programming with Non-linear constraints over the Reals

In this paper we report our effort in combining constraint logic programming with two algebraic methods for solving *non-linear* constraints: *Partial Cylindrical Algebraic Decomposition* and *Gröbner basis.* We have implemented a prototype called RISC-CLP(Real). Experience with the prototype suggests that it is desirable and in fact feasible to provide a full support of non-linear constraints. All programs are written in a portable subset of C language on top of the computer algebra C library SACLIB.

1.1 Introduction

The paradigm of constraint logic programming [35, 18, 17, 44] opened a way for intergrating logical methods and algebraic methods for problem solving. In this paper we report our effort in combining logic programming with algebraic methods for solving *non-linear* constraints over real numbers, which have been developed in computer algebra during last three decades. The algebraic methods used are: Partial Cylindrical Algebraic Decomposition [14, 6, 30] and Gröbner basis [9, 10]. We have implemented a prototype called RISC-CLP(Real).[1]

[1] RISC stands for Research Institute for Symbolic Computation.

There already exist several significant CLP languages and well-developed interpreters for them such as: Prolog III [17, 19, 18], CLP($\mathcal{R}$) [34, 25],[2] CHIP [20, 1], CAL [2, 44], etc. Among them Prolog III and CLP($\mathcal{R}$) solves constrains over real numbers.[3]

Due to the complexity of dealing with non-linear constraints over real numbers, both systems *delay* non-linear constrains, hoping that later on sufficient number of variables within them are ground so that they become linear. If such grounding does not occur, the two systems "give up" or go into an infinite loop. Further in order to reduce the complexity of basic arithmetic, CLP($\mathcal{R}$) carries out all the arithmetic computations using *floating points*, which trades accuracy with efficiency.

It is true that many interesting and real life problems can be solved without non-linear constraints or exact arithmetic. However we believe that it is also important to provide a full support of non-linearity along with exact arithmetic for the following reasons:

- *Expressive Power.*

  Non-linearity gives more expressive power to the language. There are many interesting, difficult, and important problems which cannot be solved without the full support of non-linearity, even with delay mechanism. Some of them could be solved without using non-linearity, but then the program usually becomes very *un-natural* and hard to understand.

- *Declarativeness.*

  Non-linearity gives more declarativeness to the language. In order to use the delay mechanism, one must make sure that certain non-linear constraints will be eventually made linear. This forces the programmers to think in terms of the operational semantics, and it collides with the most attractive feature of logic programming: *declarativeness.*

- *Symmetry.*

  Non-linearity gives symmetry to the language. The success or the failure of delaying depends heavily on the structure of the query. For example, $p(X, 10)$ might succeed, but $p(10, X)$ might fail. Thus the users must be careful not to make a "wrong" query on which the delay mechanism fails. This collides with another attractive feature of logic programming: *symmetry.*

- *Logical Correctness.*

[2] CLP($\mathcal{R}$) stands for the distributed version of Jaffar and Michaylov, not for the theoretical framework of Jaffar and Lassez.

[3] In fact, they solve constraints over more general domains such as tress with some leaves being reals, booleans, strings, lists, etc. But here we will only be concerned with constrains over real numbers.

Exact arithmetic gives logical correctness. Floating point computation does give efficiency, but also inaccuracy due to truncation and round-off errors. It is well known in numerical analysis that even very simple operations, when repeated, can give a totally wrong result due to accumulation of small errors. It is also very hard to predict the amount of errors a priori. This is serious especially because significant portion of constraint logic programs is made of relations over real numbers. When floating points are used in relations such as $X = Y$, one might get a wrong result: in stead of *true*, one gets *false*. This collides with the foundation of logic programming: *logic*.

For these reasons, we decided to investigate the possibility of combining the CLP interpreters with several non-linear constraint solvers along with exact arithmetic. Mainly there are two non-linear problems which need to be tackled:

- *Decision of solvability of constraints:*

  During the execution of any CLP interpreter, sets of constraints are generated to be tested for simultaneous solvability. This problem can be precisely stated as follows:

  - Given: a set $\{C_1, \ldots, C_n\}$ of constraints in variables $\{X_1, \ldots, X_m\}$.
  - Task: decide the following sentence: $(\exists X_1) \cdots (\exists X_m) C_1 \wedge \cdots \wedge C_n$.

- *Elimination of non-query variables from answer constraints:*

  In general, answer constraints contain some variables which do not appear in the query, but are generated from instantiation of program clauses. We would like to remove those "non-query" variables from the answers. This problem can be precisely stated as follows:

  - Given: a set $\{C_1, \ldots, C_n\}$ of constraints in variables $(X_1, \ldots, X_m)$ where $(X_{f+1}, \ldots, X_m)$ are non-query variables.
  - Task: find a quantifier free formula in variables $(X_1, \ldots, X_f)$, equivalent to the formula $(\exists X_{f+1}) \cdots (\exists X_m) C_1 \wedge \cdots \wedge C_n$.

In fact the first problem is a special case of the second problem, known as *quantifier elimination* in the first order theory of real closed fields. Around 1930 Tarski [46] gave the first quantifier elimination algorithm for this theory. Following Tarski's work, various improvements and new algorithms have been devised and analyzed [45, 13, 27, 8, 14, 3, 40, 5, 7, 21, 23, 22, 12, 24, 41, 42, 43, 28, 29, 38, 30, 15, 37, 11, 36, 31, 32, 33]. (See [4] for an extensive bibliography on this subject.) Among them the following three algorithms are most well-known: Collins [14], Grigor'ev [23, 22], and Renegar [41, 42, 43]. Among the three, Renegar's algorithm has the best theoretical time complexity. But as analyzed in [31], for inputs which can be decided in a tractable amount of computing time (say, less than a million years on currently available machines), Collins' algorithm is best among them.

However Collins' algorithm itself is still too slow to be used as a constraint solver in constraint logic programming. Recently in [30, 29, 15, 11, 33] the author gave a dramatic algorithmic improvement of Collins' method, obtaining speedup even up to 300,000 times; that is, a certain non-trivial problem, which would take Collins' original algorithm at least 5 months, was solved by the improved algorithm in less than 1 minute. This method is called: *Partial* Cylindrical Algebraic Decomposition (CAD) [30], and it forms the core of the constraint solver of RISC-CLP(Real).

We also used the Gröbner basis method. This method has been introduced by Buchberger [9] in 1965 in his Ph.D. dissertation, and has been improved, generalized, and applied in numerous areas, in particular, it forms a core of the powerful CLP system CAL [2, 44]. Even though Gröbner basis is a decision method over *complex numbers*, it can be sometimes used for quickly discovering *un*-solvability of constraints over real numbers, since *un*-solvability of constraints over complex numbers implies that over real numbers. It can also be used for preprocessing constraints so that the Partial CAD algorithm may go faster [11].

As now, we implemented a simple–minded CLP interpreter and combined it with the two algebraic methods mentioned above. We have not yet implemented any special algorithms for linear constraints, though we are planning to do so in the near future. All programs are written in a portable subset of C language on top of the computer algebra library SAC2 [16] which was originally written in the language ALDES [39], but translated into C language by the author.

The structure of this paper is as follows: In the following section we describe the RISC-CLP(Real) language. Then the next three sections respectively describe each component of the system: interpretor, constraint solver, and answer processor. The last section gives several program examples.

We assume that the reader is familiar with the basic terminology and concepts of constraint logic programming, though not informed about the algebraic methods mentioned above.

1.2 Language

In this section, we describe the language of RISC-CLP(Real).

1.2.1 Syntax

Currently the language of RISC-CLP(Real) has one sort, namely the sort of real numbers. The language is built up from the following symbols:

- Built-in Constant symbols: fractions such as `2/3, -1, 3.75`, etc.[4]
- Built-in Function symbols: `+ - *`
- Built-in Relation symbols: `= /= > < <= >=`
- Defined Relation symbols: alphanumeric with a first lowercase letter
- Variables: alphanumeric with a first uppercase letter
- Connectives: `:-  ,`
- Delimiters: `. ; ( )`

On top of these basic symbols, we build in the usual way the other syntactic objects such as:

- Term: is a variable, a built-in constant symbol, or of the form '$t_1 \ f \ t_2$' where t_1 and t_2 are terms, and f is a built-in function symbol.
- Atomic formula: is of the form '$r(t_1, \ldots, t_n)$' where r is a defined relation symbol with arity n, and $t_1, \ldots, t_n$ are terms.
- Constraint: is of the form '$t_1 \ r \ t_2$' where t_1 and t_2 are terms, and r is a built-in relation symbol.
- Clause: is of the form '$p \ :- \ p_1, \ldots, p_n.$' $(n \geq 0)$ where p is an atomic formula, and p_i is either an atomic formula or a constraint. We will call p the *head*, and p_i's the *body* of the clause.
- Program: is a (possibly empty) finite sequence of clauses.
- Query: is of the form '$p_1, \ldots, p_n.$', $(n \geq 1)$ where p_i is either an atomic formula or a constraint.

On top of this we add the following syntactic sugar:

- Let `U` be an arbitrary term. One can write `U^3` in stead of `U * U * U`.

Here are several example clauses:

- `disk(X,Y,X0,Y0,R) :- (X-X0)^2 + (Y-Y0)^2 <= R^2.`
- `wilkinson(X,E) :- (X+1)*(X+2)*(X+3)*(X+4) + E*X^3 = 0.`

[4] This does not mean that our domain consists of only rational numbers. It only says that we provide constant symbols for rational numbers. In principle, one could provide constant symbols for irrational numbers such as $\sqrt{2}$, but we do not see the necessity as now.

- `big_num(X) :- X >= 2^1000.`

1.2.2 Semantics

The (logical or declarative) semantics of RISC-CLP(Real) is built up from the following structure:

- Domain: The set of all real numbers.[5] Thus all variables range over real numbers.[6]
- Built-in Symbols: Mapped to the corresponding constants, functions, and relations over the set of real numbers such as 3, $\times$, $\geq$, etc.

On top of this, we build all other semantic notions such as *interpretation, solvability, model*, etc. We will not describe how they are defined here since they are done in the usual way. (For a succinct description, see [44].)

1.3 Interpretor

In this section we describe how RISC-CLP(Real) interpreter works. Figure 1.1 gives a skeletal form of the algorithm used as the interpretor. Essentially it works in the usual way: namely SLD resolution with the "left-to-right" subgoal selection strategy (though slightly modified) and the "top-to-bottom" depth-first search strategy. The differences are as follows:

- Non-linear constraints are *not* delayed. We made this "extreme" choice in this version of RISC-CLP(Real) in order to investigate the effect of delaying and non-delaying. However we believe that eventually certain compromise between the two extremes will turn out to be the best. For example, we could delay non-linear constraints only as long as there is no "sign" of infinite loop or useless backtracking.
- The left-to-right subgoal selection strategy is modified in that constraints are selected *before* atomic formulas. The motive behind this decision is to reduce the search space.
- All the constraints in a goal list are *simultaneously* sent to the constraint solver. We made this decision because we think that the solver knows more about the constraints than the interpretor and thus can make better

[5] Technically, it can be any other real closed field such as the set of all real algebraic numbers, the set of all recursive real numbers, etc.

[6] Currently we are working on another version of RISC-CLP(Real) with a more general domain such as S-expressions over real numbers.

INTERPRETOR $(C_s, C_d, A;\ u)$

Input: C_s is a set of solvable constraints.
C_d is a set of constraints.
A is a set of atomic formulas.

Output: u is '`stop`' if a solution of $C_s \cup C_d \cup A$ is found and the user wants to stop searching, '`continue`' otherwise.

Global: P is a program.

(1) [Recursion Base.]
Test the solvability of $C_s \cup C_d$ by calling the constraint solver *SOLVER* $(C_s, C_d;\ t, C)$.
If $C_s \cup C_d$ is not solvable (t is '`false`'), then set $u \leftarrow$ '`continue`' and return.
If the set A of atomic formulas is empty, then
(a) Simplify the answer C by calling the answer processor *ANSWER* $(C;\ C^*)$.
(b) Report the simplified answer C^* to the user.
(c) Ask the user if the interpreter should continue to search for another answer.
(d) If the user says 'yes', then set $u \leftarrow$'`continue`' else set $u \leftarrow$'`stop`'.
(d) Return.

(2) [Recursion.]
Remove the first atomic formula A' from A.
Let A' be of the form $r(t_1, \ldots, t_n)$.
For each clause R in P with the head relation symbol r do the following:
(a) Let R' be a variant of R, having no common variable with $A \cup \{A'\} \cup C$.
(b) Let $r(s_1, \ldots, s_n)$ be the head atomic formula of R'.
Let E be the set $\{t_1 = s_1, \ldots, t_n = s_n\}$.
Let C^* be the set of constraints in the clause R'.
Let A^* be the set of atomic formulas in the body of the clause R'.
(c) Recursively call *INTERPRETOR* $(C, E \cup C^*, A^* \cup A;\ u)$.
(d) If u is '`stop`', then return.
Set $u \leftarrow$ '`continue`'.
Return.

Figure 1.1: Algorithm: *INTERPRETOR*

decisions concerning how and in what order the constraints should be tested.

1.4 Constraint Solver

In this section, we describe how (non-linear) constraints are solved. In Figure 1.2 we give the algorithm *SOLVER*. Given a set C_s of solvable constraints and a set C_d of newly generated constraints, this algorithm decides whether the constraints in C_s and C_d are simultaneously solvable over real numbers. Logically put, it decides whether the following sentence

SOLVER $(C_s, C_d;\ t, C)$

Input: C_s is a set of solvable constraints.
C_d is a set of constraints.

Output: t is the truth of the sentence: $(\exists X_1 \in \mathbf{R}) \cdots (\exists X_m \in \mathbf{R})\ C_1 \wedge \cdots \wedge C_n$, where C_i's are the constraints in $C_s \cup C_d$ and X_i's are the variables in them.
C is a set of constraints which is equivalent to but "simpler" than $C_s \cup C_d$.

(1) [Trivial case.]
If C_d is empty, then set $C \leftarrow C_s$ and $t \leftarrow$ '`true`' and return.

(2) [Simplification.]
Set $C \longleftarrow C_s \cup C_d$.
Repeatedly apply the following transformations on C until it does not change:
- (a) Remove trivial constraints.
  (Ex: Remove constraints such as $0 = 0$, $1 > 0$, etc.)
- (b) Merge constraints with a same polynomial into one constraint.
  (Ex: $\{X^2 - 1 \leq 0, X^2 - 1 \neq 0\}$ is merged into $\{X^2 - 1 < 0\}$.)
- (c) Remove implied constraints.
  (Ex: Remove $X^2 > 0$ from $\{X^2 > 0, X^2 - 1 > 0\}$.)
- (d) Replace the equations with their reduced Gröbner basis.
  (Ex: Transform $\{XY + 1 = 0, X^2Y^4 - 1 = 0\}$ into $\{X + Y = 0, Y^2 - 1 = 0\}$.)
- (e) Reduce the non-equations with respect to the equations.
  (Ex: Transform $\{XY + 1 = 0, XY^2 > 0\}$ into $\{XY + 1 = 0, -Y > 0\}$.)

(3) [Decision.]
Let $C = (C_1, \ldots, C_n)$, and let $X_1, \ldots, X_m$ be the variables in them.
Form the sentence: $(\exists X_1 \in \mathbf{R}) \cdots (\exists X_m \in \mathbf{R})\ C_1 \wedge \cdots \wedge C_n$.
Apply the Partial CAD method on the sentence, obtaining its truth value t.
Return.

Figure 1.2: Algorithm: *SOLVER*

is a theorem of the first order theory of real closed field (FRCF):

$$(\exists X_1) \cdots (\exists X_m)\ C_1 \wedge \cdots \wedge C_n,$$

where C_i's are the constraints in $C_s \cup C_d$ and X_i's are the variables in them.

Further it produces another set C of constraints which is equivalent to but "simpler" than the set $C_s \cup C_d$. By "simpler", we roughly mean that the number of constraints is smaller, or the degrees of polynomials are smaller, the equations are triangularized, etc. It is difficult and not necessary to give an exact definition of this notion, since it is a result of various heuristics applied in the algorithm.

The algorithm *SOLVER* proceeds in two stages: *Simplification*, and *Decision*. The most significant operation during the simplification stage is the computation of *Gröbner Bases* [9, 10], and that during the decision stage is the *Partial Cylindrical Algebraic Decomposition* [30, 14].

1.4.1 Simplification

During the simplification stage, the input constraints are successively simplified by repeatedly applying various transformations on them, until no further simplification is possible. These transformations do not change the solution space of the constraints; in other words, the set of constraints before a transformation and the one after it are equivalent. Here is the list of transformations:

(a) *Remove trivial constraints.*

Quite often the constraints generated by the interpretor are of the following trivial kind: '$c\ r\ 0$', where c is a rational number and r is a relation symbol. Such constraints can be trivially decided whether they are solvable or un-solvable. If one of such constraint is un-solvable, then we know the whole set of constraints is un-solvable, thus we can immediately exit the algorithm. On the contrary, if one of such constraint is solvable, then we can simply remove it.

(b) *Merge constraints with a same polynomial into one constraint.*

We can merge several constraints involving the same polynomial into one constraint, by "collapsing" their relation symbols. For example, $\{X^2 - 1 \leq 0, X^2 - 1 \neq 0\}$ can be merged into $\{X^2 - 1 < 0\}$. As a result, the polynomials occurring in the constraints are distinct. In fact, all the polynomials in the constraints are first made monic before this transformation is applied, so that we can merge even the set such as $\{2X^2 - 2 \leq 0, 3X^2 - 3 \neq 0\}$ into $\{X^2 - 1 < 0\}$.

(c) *Remove implied constraints.*

A constraint is implied by the other constraints if the solution space of the constraint contains the solution space of the other constraints. For example, let $P(\vec{X}) > 0$ and $Q(\vec{X}) > 0$ be two constraints. Let $\hat{F}$ be the sentence: $(\forall \vec{X}) P(\vec{X}) > 0 \Longrightarrow Q(\vec{X}) > 0$. If this sentence is true, then $P(\vec{X}) > 0$ implies $Q(\vec{X}) > 0$, and thus we can safely remove the constraint $Q(\vec{X}) > 0$. Though this check is possible in principle, it can be time-consuming. Thus we instead test a stronger sentence: $(\forall \vec{X}) Q(\vec{X}) - P(\vec{X}) > 0$. But even this test is time-consuming in full generality. Thus currently we only test whether $Q(\vec{X}) - P(\vec{X})$ is a positive rational number.

(d) *Replace the equations with their reduced Gröbner basis.*

It is beyond the scope of this paper to give an exposition of Gröbner basis method, and we only informally point out several features which are useful in the context of constraint solving over real numbers. For a detailed exposition, see [10]. Given a set E of polynomials, the

Gröbner basis algorithm produces a "canonical" set G of polynomials which generates the same ideal as the input set. In particular the set of common complex solutions of the polynomials E is the same as that of G. Furthermore, the canonical set G is "triangularized" when constructed with pure-lexicographic ordering. (For details, see [10].) This can be in fact viewed as a generalization of Gaussian elimination for a system of non-linear polynomial equations. This can be also viewed as a generalization of "linearization" of non-linear constraints by using grounded variables. This triangularization can also help in speeding up the execution of the Partial CAD algorithm [11].

Most importantly, the computation of Gröbner bases can sometimes help in quickly discovering un-solvability of constraints over real numbers. As an example, consider the following set of constraints:

$$\{XY - 1 = 0, XY^2 = 0, X^2 > Y\}$$

The reduced Göbner basis of the first two polynomials $\{XY-1, XY^2\}$ is just $\{1\}$. Thus the above constraints are transformed into the following:

$$\{1 = 0, X^2 > Y\}$$

which is trivially un-solvable, even over real numbers.

In fact we have the following theorem by Buchberger:

> *Let E be a set of multivariate polynomials with rational number coefficients. Let G be the reduced Gröbner basis of E. Then the system of polynomial equations made of the polynomials in E is un-solvable over complex numbers iff $G = \{1\}$.*

This theorem guarantees that we can always decide un-solvability of a system of polynomial equations over complex numbers by simply checking if its reduced Gröbner basis is $\{1\}$. Since un-solvability of a system of polynomial equations over complex numbers implies that over real numbers, this method can be often used for quickly discovering un-solvability of constraints over real numbers.

(e) *Reduce the non-equations with respect to the equations.*

We can "reduce" the polynomials in the non-equations by using the polynomials in the equations. For example, consider the following set of constraints: $\{P(\vec{X}) = 0, Q(\vec{X}) > 0\}$. We can replace $Q(\vec{X}) > 0$ by $Q(\vec{X}) - M(\vec{X})P(\vec{X}) > 0$, where $M(\vec{X})$ is any polynomial. Obviously the polynomial M is chosen such that the resulting inequality is simpler than the original one. Such a polynomial M can be computed by a "generalized polynomial division". In fact, this operation is routinely carried out during Gröbner basis computation in order to simplify intermediate polynomials which are generated.

1.4.2 Decision

When all the above transformations do not simplify the set C of constraints any more, we finally use the Partial CAD method to decide the solvability of the set C. Specifically, let $C = (C_1, \ldots, C_n)$. We form the sentence:

$$(\exists X_1 \in \mathbf{R}) \cdots (\exists X_m \in \mathbf{R}) \;\; C_1 \wedge \cdots \wedge C_n$$

where $X_1, \ldots, X_m$ are the variables occurring in $C_1, \ldots, C_n$. Then we apply the Partial CAD method on the sentence, obtaining its truth value.

It is beyond the scope of this paper to give a through exposition of the Partial CAD method, and we merely give a brief informal description of the main ideas. For a detailed exposition, see [30, 15, 6, 14]. As mentioned in the introduction, the Partial CAD method can solve a more general problem than the decision of existential sentences such as the above; namely quantifier elimination problem in the first order theory of real closed fields. Though constraint solving does not require the full power of quantifier elimination, simplification of answers does require quantifier elimination (as will be shown in the next section). Therefore here we will describe the Partial CAD method in the context of full generality of quantifier elimination.

Let $\hat{F} = (Q_{f+1}X_{f+1}) \cdots (Q_m X_m) F(X_1, \ldots, X_m)$ be a prenex formula in the first order theory of real closed fields where Q_i are quantifiers and F is a quantifier-free formula. The method begins by constructing a certain finite decomposition of $\mathbf{R}^m$ such that in each element (cell) of the decomposition, the polynomials occurring in the formula F have constant signs. Thus the signs of the polynomials on a cell can be determined by evaluating the polynomials on a "sample" point belonging to the cell. Such a decomposition is called *Cylindrical Algebraic Decomposition* (CAD). A CAD is constructed in three stages:

- *Projection Stage:*

  The input m–variate polynomials are "projected" into a set of $(m-1)$–variate polynomials in a way that a CAD of m–space can be built on top of a CAD of $(m-1)$–space. This process is successively carried out until univariate projection polynomials are obtained.

- *Base Stage:*

  A CAD of 1–space is constructed through isolating the real roots of the univariate projection polynomials.

- *Extension Stage:*

  A CAD of 2–space is constructed by building "stacks" of cells over the cells of the CAD of 1–space. This process is successively carried out until a CAD of m–space is constructed.

Once a CAD of the polynomials in F is constructed, the truth value of F on each cell can be decided by (1) evaluating the signs of the polynomials on the cell, (2) determining the truth values of the atomic formulas of F on the cell, using the signs of the polynomials and the relation symbols in the atomic formulas, and (3) finally inferring the truth value of F on the cell from the truth values of its atomic constituents.

Once the truth values of F on the cells of the CAD of $\mathbf{R}^m$ are determined, the method infers from them the truth values of $(Q_m X_m)F$ on the cells of the CAD of $\mathbf{R}^{m-1}$. Continuing in the same manner, the truth values of $(Q_{f+1}X_{f+1})\cdots(Q_m X_m)F$ on the CAD of $\mathbf{R}^f$ are determined. Now the union of the "true" cells of the CAD of $\mathbf{R}^f$ forms the "solution set" of the input formula. Finally the method construct a quantifier free formula defining the solution set, by forming a disjunction of defining formulas for each true cell.

The improvements introduced by the author on this method mainly consists of the followings:

- *Improvements in the projection.*

  By generalizing a lemma on which the proof of the original projection operation is based, we are able to reduce the size of the projection set significantly, and thus speed up the whole quantifier elimination process [30, 29]

- *Improvements in the extension.*

  By intermingling the extension with the truth evaluation, we are usually able to complete quantifier elimination with only a partially built CAD, resulting in significant speedup [30, 15]

- *Improvements in the solution formula construction.*

  By constructing defining formulas for a collection of cells instead of individual cells and also by using 3-valued logic minimization, we are often able to produce much simpler solution formulas, without incurring the enormous cost of augmented projection. [30, 33].

As can be seen from the fact that we form a union $C = C_s \cup C_d$ at the beginning of the simplification stage of the algorithm *SOLVER*, the Partial CAD method has not yet been made incremental. We believe that this is possible, though it seems to require significant restructuring of the method. Various possibilities are being pursued, and the details will be published in the near future.

1.5 Answer Processor

In this section, we describe the answer processor. The main tasks of this component are *removal of non-query variables* from answer constraints and their *simplification*.

1.5.1 Removal of non-query variables

Before we give a precise statement of the task, we first give a motivating example. Consider the following program and the query:

```
resistor(V,I,R) :-
        V = I*R, R > 0.

par_circuit(V,I,R1,R2) :-
        I = I1 + I2,
        resistor(V,I1,R1),
        resistor(V,I2,R2).

?- par_circuit(V,I,R1,R2).
```

Upon this query, RISC-CLP(Real) first generates the following solvable set of constraints:

```
I2.2.2 + I1.2.1 - I = 0
R2*I1.2.1 - I*R2 + V = 0
R1*I1.2.1 - V = 0
I*R1*R2 - V*R2 - V*R1 = 0
R2 > 0
R1 > 0
```

This set of constrains is expressed in the following variables:

V, I, R1, R2, I1.2.1, I2.2.2

Among them, the two variables `I1.2.1` and `I2.2.2` are not present in the query, but are generated during an instantiation of the second clause of the program. Now we would like to remove these "non-query variables" from the constraints so that the resulting constraints are expressed only in the "query variables". Here is such a desired set of constraints:

```
R1 > 0
R2 > 0
I*R1*R2 - V*R2 - V*R1 = 0
```

The relationship between these two sets of constraints can be roughly described as follows: the "solution space" of the second set of constraints is the "projection" of that of the first set of constraints with respect to the query-variables. In order to make this notion more precise, we first give two definitions:

- Let C be a set of constraints expressed in variables $(X_1, \ldots, X_m)$. The *solution space* of C is the set $\{P \in \mathbf{R}^m \mid P \text{ satisfies } C\}$.
- Let S be any subset of $\mathbf{R}^m$, and let $0 < f < m$. The *projection* of S on $\mathbf{R}^f$ is the set $\{(p_1, \ldots, p_f) \in \mathbf{R}^f \mid \text{ there exists } (p_{f+1}, \ldots, p_m) \in \mathbf{R}^{m-f} \text{ such that } (p_1, \ldots, p_m) \in S\}$.

Now we are ready to give a precise description of the relationship between the constraints with non-query variables and those without them: Let C be a set of constraints with variables $(X_1, \ldots, X_m)$ where $(X_{f+1}, \ldots, X_m)$ are non-query variables. Let C' be a desired set of constraints with only the variables $(X_1, \ldots, X_f)$. Then the solution space of C' is the projection of the solution space of C on $\mathbf{R}^f$. Thus our task can now be precisely described as

- Given: a set C of constraints in variables $(X_1, \ldots, X_m)$ where the variables $(X_{f+1}, \ldots, X_m)$ are non-query variables.
- Find: a set C' of constraints in variables $(X_1, \ldots, X_f)$ such that the solution space of C' is the projection of that of C on $\mathbf{R}^f$.

Using the definitions of solution space and projection, we can rewrite the above task description as follows:

- Given: a set $\{C_1, \ldots, C_n\}$ of constraints in the variables $(X_1, \ldots, X_m)$ where $(X_{f+1}, \ldots, X_m)$ are non-query variables.
- Find: a quantifier free formula in variables $(X_1, \ldots, X_f)$, equivalent to the formula $(\exists X_{f+1}) \cdots (\exists X_m) C_1 \wedge \cdots \wedge C_n$.

Note that we try to find *quantifier free formula*, which is more general form than a set (conjunction) of constraints. It is because sometimes it is not possible to find a conjunction of constraints which satisfy the above condition.

Once the problem is stated in this way, one sees immediately that the Partial CAD method can solve this problem. One simply needs to input the formula

$$(\exists X_{f+1}) \cdots (\exists X_m) C_1 \wedge \cdots \wedge C_n$$

to the Partial CAD algorithm, which outputs a desired quantifier free formula. In fact, RISC-CLP(Real) eliminates non-query variables exactly in this way.

1.5.2 Simplification of Answers

The quantifier free formulas generated by the Partial CAD algorithm are not necessarily what the human user might prefer to see. First of all, the quantifier free formulas are not necessarily in the usual disjunctive normal form. Secondly, though the Partial CAD algorithm carries out simplification of formulas on the basis of logical connectives and relation symbols, it does not do so on the basis of polynomial strucutres. For example, the formula $X - 1 = 0 \wedge XY + 1 > 0$ is not simplified to $X = 1 \wedge Y > 1$. Thus before an answer is reported to the user, such simplifications are carried out by the answer processor. This is done as follows:

- First, the quantifier free formula is rewritten as a disjunction of conjunctions of constraints.
- Second, the simplification operations analogous to the ones used in the algorithm *SOLVER* are applied to each conjunction of constraints.
- Third, every conjunction (of constraints) which is implied by the other conjunctions is removed.

1.6 Program Examples

In this section, we give several simple examples, comparing RISC-CLP(Real) and CLP($\mathcal{R}$) on them. The purpose of such comparison is not to give an evaluation of RISC-CLP(Real) over CLP($\mathcal{R}$), rather to illustrate the need of full support of non-linear constraint solving.

1.6.1 Electric Circuit [34, 26]

The following program describes a simple circuit made of one battery V and two parallel resistors R_1 and R_2.

```
resistor(V,I,R) :-
        V = I*R, R > 0.

par_circuit(V,I,R1,R2) :-
        I = I1 + I2,
        resistor(V,I1,R1),
        resistor(V,I2,R2).
```

The following query asks for the relationship between V, I, R_1 and R_2:

```
?- par_circuit(V,I,R1,R2).
```

Upon this query, RISC-CLP(Real) answers as following:

```
R1 > 0
R2 > 0
I*R1*R2 - V*R2 - V*R1 = 0
```

Upon the same query, CLP($\mathcal{R}$) answers as following:

```
0 < R1
0 < R2
V = _t1 * R2
V = (I - _t1) * R1
*** Maybe
```

Note that CLP($\mathcal{R}$) returns the answer which still contains a non-query variable "`_t1`", and it also says "**`Maybe`**", indicating that it cannot tell whether the non-linear constraints are in fact solvable.

The next query asks if it is possible to obtain a current $I > 3$ with the battery $V = 10$ and resistors $R_1 > 5$ and $R_2 > 10$:

```
?- par_circuit(10,I,R1,R2), R1 > 5, R2 > 10, I > 3.
```

Upon this query, RISC-CLP(Real) answers:

```
*** No.
```

But CLP($\mathcal{R}$) answers:

```
5 < R1
10 < R2
3 < I
10 = _t2 * R2
10 = (I - _t2) * R1
*** Maybe
```

Now the last query asks if it is possible to obtain a current $I > 2.99999999$ with $V = 10$, $R_1 = 5$, and $R_2 = 10$, and if so find one.

```
?- par_circuit(10,I,5,10), I > 2.99999999.
```

Upon this query, RISC-CLP(Real) answers:

```
I = 3
```

But CLP($\mathcal{R}$) answers:

```
*** No
```

CLP($\mathcal{R}$) returns this wrong answer because it first rounds up the number 2.99999999 to 3, then checks if $3 > 3$.

1.6.2 Complex Number Arithmetic [34]

The following program defines the multiplication of two complex numbers $R_1 + iI_1$ and $R_2 + iI_2$ into $R_3 + iI_3$.

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

The following query asks for $(1 + 2i)(3 + 4i)$.

```
?- zmul(1,2,3,4,R3,I3).
```

Both RISC-CLP(Real) and CLP($\mathcal{R}$) answer as:

```
R3 = -5
I3 = 10
```

Now we make another query asking for R_1 and R_2, while fixing the values of the other variables as in the previous query and its answer:

```
?- zmul(R1,2,R2,4,-5,10).
```

Upon this query, RISC-CLP(Real) answers

```
R1 = 1
R2 = 3

OR

R1 = 3/2
R2 = 2
```

It found the expected answer $R_1 = 1$, $R_2 = 3$, but it also found another answer $R_1 = 3/2$, $R_2 = 2$. In fact it is easy to verify that $(3/2+2i)(2+4i) = (-5 + 10i)$. On the same query, CLP($\mathcal{R}$) answers:

```
R1 = -0.5*R2 + 2.5
3 = R1 * R2
*** Maybe
```

The next query asks for a square root of $-3 + 4i$.

```
?- zmul(X,Y,X,Y,-3,4).
```

RISC-CLP(Real) answers:

```
X = 1
Y = 2

OR

X = -1
Y = -2
```

CLP($\mathcal{R}$) answers:

```
-X * Y + 4 = X * Y
Y * Y - 3 = X * X
*** Maybe
```

1.6.3 Mortgage [25, 34]

The following program relates some simple parameters in a mortgage.

```
mortgage(P,T,I,B,M) :-
        T = 1,
        B = P*(1 + I) - M.
mortgage(P,T,I,B,M) :-
        T > 1,
        mortgage(P*(1 + I) - M, T - 1, I, B, M).
```

In the above, the parameters stand for the principal, number of months, interest rate, outstanding balance, and monthly payment. Note that this program, unlike the previous ones, uses recursion.

Consider the following situation. You are a banker. A customer demands to borrow 100,000 dollars from you and insists that he will not pay more than 25,000 dollars per month, and that not more than five times. As a banker, you want to get at least 10% of interest. The following query asks whether it is possible to satisfy both parties.

```
?- mortgage (100000,T,I,0,M), T <= 5, M <= 25000, I >= 0.1.
```

RISC-CLP(Real) answers:

```
*** No.
```

To the same query, CLP($\mathcal{R}$) answers:

```
T = 2
M <= 25000
0.1 <= I
M = (-M + 100000*I + 100000) * (I + 1)
```

```
*** Retry? y

T = 3
M <= 25000
0.1 <= I
M = _t12 * (I + 1)
_t12 + M = (-M + 100000*I + 100000) * (I + 1)

*** Retry? y

T = 4
M <= 25000
0.1 <= I
M = _t16 * (I + 1)
_t16 + M = _t12 * (I + 1)
_t12 + M = (-M + 100000*I + 100000) * (I + 1)

*** Retry? y

T = 5
M <= 25000
0.1 <= I
M = _t20 * (I + 1)
_t20 + M = _t16 * (I + 1)
_t16 + M = _t12 * (I + 1)
_t12 + M = (-M + 100000*I + 100000) * (I + 1)

*** Retry? y
```

After the last retry, it goes into an infinite loop. If the constraints in the query are moved to the beginning, it will produce the same answers, except that the last retry will result in '`No`' answer.

1.6.4 Pythagorean Numbers

The following program is designed to find three natural numbers (positive integers) X, Y, Z such that $X^2 + Y^2 = Z^2$.

```
nat(X) :-
        X = 1.
nat(X) :-
        X > 1, nat(X-1).

pyth(X,Y,Z) :-
        X*X + Y*Y = Z*Z, X <= Y, nat(Z), nat(X), nat(Y).
```

We ask the following query:

```
?- pyth(X,Y,Z).
```

Then RISC-CLP(Real) answers:

```
X = 3
Y = 4
Z = 5

*** Retry? y

X = 6
Y = 8
Z = 10

*** Retry? y

X = 5
Y = 12
Z = 13

*** Retry? n
```

If one would respond `y` to the third retry prompt, it would find the next values for X, Y, and Z, and so on. On the same query, CLP($\mathcal{R}$) goes immediately into an *infinite loop*. In order to keep it from inifite loop, one might have to add another constraint such as '`Y <= Z`' to the last clause of the program. But for the queries such as '`pyth(1,1,Z)`', the modification would not help: CLP($\mathcal{R}$) will still go into an infinite loop, while RISC-CLP will answer '`No`' (independent of the modification).

1.6.5 Wilkinson Polynomial

From now on, we will only show the sessions with RISC-CLP(Real), since all the following problems are beyond the power of CLP($\mathcal{R}$). The following program describes the famous Wilkinson polynomial equation [47]

$$\prod_{i=1}^{20}(X+i) + EX^{19} = 0$$

```
wilkinson(X,E) :-
        (X+1)*(X+2)*(X+3)*(X+4)*(X+5)*
        (X+6)*(X+7)*(X+8)*(X+9)*(X+10)*
        (X+11)*(X+12)*(X+13)*(X+14)*(X+15)*
        (X+16)*(X+17)*(X+18)*(X+19)*(X+20) + E*X^19 = 0.
```

When expanded, the above polynomial equation is as follows:

```
X^20 +
210*X^19 + E*X^19
20615*X^18 +
1256850*X^17 +
53327946*X^16 +
1672280820*X^15 +
40171771630*X^14 +
756111184500*X^13 +
11310276995381*X^12 +
135585182899530*X^11 +
1307535010540395*X^10 +
10142299865511450*X^9 +
63030812099294896*X^8 +
311333643161390640*X^7 +
1206647803780373360*X^6 +
3599979517947607200*X^5 +
8037811822645051776*X^4 +
12870931245150988800*X^3 +
13803759753640704000*X^2 +
8752948036761600000*X +
2432902008176640000  = 0
```

The following easy query asks for the real roots of the unperturbed polynomial ($E = 0$) between the closed interval $[-20, -10]$.

```
wilkinson(X,0), -20 <= X, X <= -10.
```

Then RISC-CLP(Real), as expected, lists all the eleven real roots:

```
X = -20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10.
```

Now we perturb the polynomial by an extremely small amount $E = 2^{-23}$, and make the same query as before:

```
wilkinson(X,(1/2)^23), -20 <= X, X <= -10.
```

Then RISC-CLP(Real) surprises you by the answer:

```
*** No.
```

Yes, all the eleven real roots in $[-20, -10]$ disappeared! And this is why it is famous. It vividly illustrates that even extremely small error introduced by floating point computation can dramatically change the root behavior.

1.6.6 Polynomial Real Roots

The following query asks for the condition on the coefficients of an (at most) cubic polynomial to have a real root:

```
?- A*X^3 + B*X^2 + C*X + D = 0 {A,B,C,D}.
```

Note that RISC-CLP(Real) allows one to specify the variables which can appear in answers, by enclosing them between a pair of braces. We obtain the following answer:

```
A /= 0

OR

B*D - 1/4*C^2 <= 0
A*C - 8/27*B^2 /= 0

OR

D = 0

OR

B*D - 1/4*C^2 < 0
```

The next query asks for the condition on P, Q, R such that the polynomial $X^4 + PX^2 + QX + R$ has a real root.

```
?- X^4 + P*X^2 + Q*X + R = 0 {P,Q,R}.
```

We obtain the following answer:

```
P*R - 9/8*Q^2 - 1/4*P^3 > 0
Q^2 + 8/27*P^3 <= 0

OR

P*R - 9/8*Q^2 - 1/4*P^3 /= 0
R^3 - 1/2*P^2*R^2 + 9/16*P*Q^2*R + 1/16*P^4*R - 27/256*Q^4 - 1/64*P^3*Q^2 <= 0

OR

R^3 - 1/2*P^2*R^2 + 9/16*P*Q^2*R + 1/16*P^4*R - 27/256*Q^4 - 1/64*P^3*Q^2 <= 0
P <= 0
```

1.6.7 Catching a Bird with a Sling

You are out in the field with a sling, hoping to catch some birds. Now you see a bird flying right above your head, and you immediately begin to shoot several stones one after the other. Obviously, you need to choose a right angle for shooting. The following program captures the scenario:

```
%-------------------------------------------------------------
% A bird is modeled as a horizontal line segment of length L.
% Its left end point is initially located at (0,H).
% The point (X,Y) is in the bird initially.

bird_init(X,Y,H,L) :-
        X >= 0,
        X <= L,
        Y = H.

%-------------------------------------------------------------
% The bird moves horizontally to the right with the speed U.
% (Dx,Dy) is the displacement of the bird at time T.

bird_displacement(T,Dx,Dy,U) :-
        Dx = U*T,
        Dy = 0.

%--------------------------------------------------------------
% (X,Y) is in the bird at time T.

bird(T,X,Y,H,L,U) :-
        bird_displacement(T,Dx,Dy,U),
        bird_init(X-Dx,Y-Dy,H,L).

%-------------------------------------------------------------
% A stone is modeled as a point.
% The (first) stone is initially located at (0,0).

stone_init(X,Y) :-
        X = 0,
        Y = 0.

%-------------------------------------------------------------
% The stones are shot with the velocity (Vx,Vy).
% The gravitational acceleration on the earth is -9.8 meter/sec^2.
% (Dx,Dy) is the displacement of the (first) stone at time T.

stone_displacement(T,Dx,Dy,Vx,Vy) :-
```

```
        Dx = Vx*T,
        Dy = Vy*T - 1/2*9.8*T^2.

%-------------------------------------------------------------
% (X,Y) is in the (first) stone at time T.

stone(T,X,Y,Vx,Vy) :-
        stone_displacement(T,Dx,Dy,Vx,Vy),
        stone_init(X-Dx,Y-Dy).

%-------------------------------------------------------------
% X is a positive integer.

nat(X) :-
        X = 1.
nat(X) :-
        X > 1, nat(X-1).

%-------------------------------------------------------------
% The stones are shot at time intervals of I.
% (X,Y) is in the N-th stone at time T.

stone_N_th(T,X,Y,Vx,Vy,I,N) :-
        nat(N),
        stone(T-I*(N-1),X,Y,Vx,Vy).

%-------------------------------------------------------------
% V is the speed of shooting:  V^2 = Vx^2 + Vy^2.
% S is the slope of shooting:  S = Vy/Vx.
% The N-th stone hits the bird.

hit(H,L,U,V,S,I,N) :-
        V^2 = Vx^2 + Vy^2,
        Vy = S*Vx,
        S >= 0,
        Vx >= 0,
        Vy >= 0,
        T >= 0,
        X >= 0,
        Y >= 0,
        bird(T,X,Y,H,L,U),
        stone_N_th(T,X,Y,Vx,Vy,I,N).

%-------------------------------------------------------------
```

The following query describes the situation where a bird of size 0.2 meter is flying on 20 meters high, with speed 10 meters per second. You shoot 3 stones with speed 30 meters per second, in time intervals of 2 seconds, with slopes not greater than 45°. Now the goal is to find the condition on the slope of shooting such that one of the stones hits the bird, and also to find out which stone it is.

```
hit(20,0.2,10,30,S,2,N), N <= 3, S <= 1.
```

Upon this query, RISC-CLP(Real) answers as following:

```
S <= 1
S >= 0
S^4 - 0.00105948*S^3 - 15.7155*S^2 + 31.573*S - 16.4013 >= 0
S^4 - 0.0708717*S^3 - 15.6471*S^2 + 31.8214*S - 16.6471 <= 0
N = 2

*** Retry? y
*** No.
```

Thus it tells you that no matter what slope of shooting you choose, the *first* and the *third* stones can *never* hit the bird. But with a proper slope, the *second* stone *can* hit the bird. If one is interested in obtaining an approximate interval for a proper slope, one can use any computer algebra package (which we plan to combine with RISC-CLP(Real) in the very near future) to solve the above answer constraint, finding that

$$0.9098202541 < S < 0.9123792466$$

$$42.29656911° < \arctan(S) < 42.37668458°$$

which tells you how hard it is to catch a bird with a sling!

1.7 Conclusion

In this paper, we reported our effort in combining constraint logic programming with two algebraic methods for solving *non-linear* constraints over real numbers, which have been developed in computer algebra during last three decades: namely, the method of Partial Cylindrical Algebraic Decomposition, and the method of Gröbner basis.

Experience with the prototype system RISC-CLP(Real) suggest that it is desirable and in fact feasible to provide a full support of non-linear constraints. We hope that non-linear constraint solvers are systematically investigated and optimized as linear constraint solvers have been.

I would like to thank Bruno Buchberger and Alain Colmerauer for introducing me to this exciting area of constraint logic programming! I am also grateful to Joxan Jaffar and his research group for making avaiable of the excellent interpreter CLP($\mathcal{R}$) free of charge. Without their contribution, the section on the program examples in this paper was not possible. Wolfgang Windsteiger implemented the Gröbner basis algorithm and Olga Caprotti helped me in preparing the draft of this paper and implementing some part of the interpretor.

Bibliography

[1] A. Aggoun, M. Dincbas, A. Herold, H. Simonis, and P. Van Hentenryck. The CHIP System. Technical Report TR-LP-24, European Computer Industry Research Centre (ECRC), Munich, Germany, June 1987.

[2] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88), ICOT, Tokyo*, pages 263–276, December 1988.

[3] D. S. Arnon. Algorithms for the geometry of semi-algebraic sets. Technical Report 436, Computer Sciences Dept, Univ. of Wisconsin-Madison, 1981. Ph.D. Thesis.

[4] D. S. Arnon. A bibliography of quantifier elimination for real closed fields. *Journal of Symbolic Computation*, 5(1,2):267–274, 1988.

[5] D. S. Arnon. A cluster-based cylindrical algebraic decomposition algorithm. *Journal of Symbolic Computation*, 5(1,2):189–212, 1988.

[6] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: The basic algorithm. *SIAM J. Comp.*, 13:865–877, 1984.

[7] M. Ben-Or, D. Kozen, and J. H. Reif. The complexity of elementary algebra and geometry. *J. Comput. System Sci.*, 32(2):251–264, 1986.

[8] W. Böge. Decision procedures and quantifier elimination for elementary real algebra and parametric polynomial nonlinear optimization. Manuscript in preparation, 1980.

[9] B. Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal.* PhD thesis, Universitat Innsbruck, Institut fur Mathematik, 1965. German.

[10] B. Buchberger. Groebner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Recent Trends in Multidimensional Systems Theory*, chapter 6. D. Riedel Publ. Comp., 1983.

[11] B. Buchberger and H. Hong. Speeding-up quantifier elimination by Groebner bases. Technical Report 91-06.0, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1991.

[12] J. Canny. Some algebraic and geometric computations in PSPACE. In *Proceedings of the 20th annual ACM symposium on the theory of computing*, pages 460–467, 1988.

[13] P. J. Cohen. Decision procedures for real and p-adic fields. *Comm. Pure and Applied Math.*, 22:131–151, 1969.

[14] G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Lecture Notes In Computer Science*, pages 134–183. Springer-Verlag, Berlin, 1975. Vol. 33.

[15] G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, September 1991.

[16] G. E. Collins and R. Loos. *The SAC-2 Computer Algebra System*. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria A-4040.

[17] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[18] Alain Colmerauer. Opening the prolog-III universe. *BYTE Magazine*, 12(9), August 1987.

[19] Alain Colmerauer. Final specifications for prolog-III. Technical Report P1219(1106), ESPRIT, February 1988.

[20] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, December 1988.

[21] N. Fitchas, A. Galligo, and J. Morgenstern. Algorithmes repides en séquential et en parallele pour l'élimination de quantificateurs en géométrie élémentaire. Technical report, UER de Mathématiques

Universite de Paris VII, 1987. To appear in: Séminaire Structures Algébriques Ordonnées.

[22] D. Yu. Grigor'ev. The complexity of deciding Tarski algebra. *Journal of Symbolic Computation*, 5(1,2):65–108, 1988.

[23] D. Yu. Grigor'ev and N. N. Vorobjov (Jr). Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5(1,2):37–64, 1988.

[24] J. Heintz, M-F. Roy, and P. Solernó. On the complexity of semialgebraic sets. In *Proc. IFIP*, pages 293–298, 1989.

[25] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. *The CLP(R) Programmer's Manual Version 1.1*, November 1991.

[26] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 675–703. MIT Press, May 1987.

[27] C. Holthusen. *Vereinfachungen für Tarski's Entscheidungsverfahren der elementaren reellen Algebra.* PhD thesis, University of Heidelberg, January 1974.

[28] H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. Technical Report OSU-CISRC-12/89 TR55, Computer Science Dept, The Ohio State University, 1989.

[29] H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *International Symposium of Symbolic and Algebraic Computation*, pages 261–264, 1990.

[30] H. Hong. *Improvements in CAD–based Quantifier Elimination.* PhD thesis, The Ohio State University, 1990.

[31] H. Hong. Comparison of several decision algorithms for the existential theory of the reals. Technical Report 91–41.0, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1991.

[32] H. Hong. Parallelization of quantifier elimination on workstation network. Technical Report 91-55.0, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1991.

[33] H. Hong. Simple solution formula construction in cylindrical algebraic decomposition based quantifier elimination. Technical Report 92-02.0, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1992. To appear in the proceedings of *International Conference on Symbolic and Algebraic Computation* ISSAC-92.

[34] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, Proceedings 4^{th} ICLP, pages 196–218, Cambridge, MA, May 1987. The MIT Press.

[35] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[36] J. R. Johnson. *Algorithms for Polynomial Real Root Isolation.* PhD thesis, The Ohio State University, 1991.

[37] L. Langemyr. The cylindrical algebraic decomposition algorithm and multiple algebraic extensions. In *Proc. 9th IMA Conference on the Mathematics of Surfaces*, September 1990.

[38] D. Lazard. An improved projection for cylindrical algebraic decomposition. Unpublished manuscript, 1990.

[39] R. G. K. Loos. The algorithm description language ALDES (Report). *ACM SIGSAM Bull.*, 10(1):15–39, 1976.

[40] S. McCallum. *An Improved Projection Operator for Cylindrical Algebraic Decomposition.* PhD thesis, University of Wisconsin-Madison, 1984.

[41] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals (part I). Technical Report 853, Cornell University, Ithaca, New York 14853-7501 USA, July 1989.

[42] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals (part II). Technical Report 854, Cornell University, Ithaca, New York 14853-7501 USA, July 1989.

[43] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals (part III). Technical Report 856, Cornell University, Ithaca, New York 14853-7501 USA, August 1989.

[44] K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Applications. *Journal of Symbolic Computation*, 8:589–603, 1989.

[45] A. Seidenberg. A new decision method for elementary algebra. *Ann. of Math*, 60:365–374, 1954.

[46] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. Univ. of California Press, Berkeley, second edition, 1951.

[47] J. H. Wilkinson. The evaluation of the zeros of ill-conditioned polynomials. parts i and ii. *Numerische Mathematik*, 1:150–166,167–180, 1959.