Quantifier Elimination-Based Constraint Logic Programming

Thomas Sturm*
FMI, University of Passau, Germany

MIP-0202
January 22, 2002

Abstract

We present an extension of constraint logic programming, where the admissible constraints are arbitrary first-order formulas over some domain. Constraint solving is realized by effective quantifier elimination. The current realization in our CLP(RL) system provides as possible domains $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{Q}_p$ for primes p. In $\mathbb{R}$ and $\mathbb{C}$ we admit arbitrary degrees in our constraints. In $\mathbb{Q}_p$ we restrict to linear formulas. The arithmetic is always exact. We describe the conceptual advantages of our approach and the capabilities of CLP(RL).

*`sturm@uni-passau.de`, `http://www.fmi.uni-passau.de/~sturm/`

1 Introduction

Logical programming languages have emerged during the early seventies with Prolog by Colmerauer and Kowalski being the by far most prominent example. The first version of Prolog has been finished in 1973 [CR93]. The first efficient implementation in form of a Prolog compiler has been finished by Warren in 1977 [War77]. The major conceptual contribution, as stated by Kowalski [Kow79], was disconnecting *logic* from *control.* The programmer should not longer be concerned with specifying and coding algorithmic control structures but instead *declaratively* specify the problem to be solved within some formal logical framework. The programming language itself would then contain a universal control algorithm for solving the specified problems.

The logical framework was universal Horn clauses. The universal control algorithm was a deterministic variant of J. A. Robinson's resolution principle [Rob65] for automatic theorem proving. Prolog became surprisingly successful, in particular in connection with the Japanese 5th Generation Computing Project. It had, however, turned out that the pure approach of declarative specification resulted in a lack of efficiency that was not acceptable. This particularly affected arithmetic, which had to be defined inductively using a successor function.

Let us have a look at the usual logic programming solution for addition on natural numbers:

$$\begin{array}{rcl}
\mathrm{nat}(0) & \leftarrow & \\
\mathrm{nat}\big((s(X)\big) & \leftarrow & \mathrm{nat}(X) \\
\mathrm{plus}(X, 0, X) & \leftarrow & \mathrm{nat}(X) \\
\mathrm{plus}\big(X, s(Y), s(Z)\big) & \leftarrow & \mathrm{plus}(X, Y, Z).
\end{array}$$

The execution of such a program consists in stating *queries* like

$$\leftarrow \mathrm{plus}\big(s(s(s(0))), s(0), Z\big),$$

which would by means of resolution result in the *answer* $Z = s(s(s(s(0))))$.

For introducing some notions on one hand and reminding the reader of the common procedure on the other hand, we sketch how such a query is processed by SLD resolution. The *body* $\mathrm{plus}\big(s(s(s(0))), s(0), Z\big)$ of our query forms the initial *goal*

$$G = \big\{\mathrm{plus}(s(s(s(0))), s(0), Z)\big\}.$$

The *interpreter* matches via unification the only contained *atom* with the *head* $\mathrm{plus}\big(X_1, s(Y_1), s(Z_1)\big)$ of the *instance*

$$\mathrm{plus}\big(X_1, s(Y_1), s(Z_1)\big) \leftarrow \mathrm{plus}(X_1, Y_1, Z_1)$$

of the last clause of the program. This creation of instances by giving fresh names to all variables is called *standardizing apart.* A suitable *most general unifier* (MGU) for this is

$$\mu' = \mu_0 = \left\{X_1 = s(s(s(0))), Y_1 = 0, Z = s(Z_1)\right\}.$$

The selected atom in the initial goal is now replaced by the atoms obtained from the body of our considered clause instance by applying the MGU:

$$G = \left\{\text{plus}(s(s(s(0))), 0, Z_1)\right\}.$$

At this point, we remove from μ' all bindings for variables that occur neither in the initial query nor in the current goal. The remaining new MGU is $\mu = \left\{Z = s(Z_1)\right\}$. The first and only atom in G matches the head of

$$\text{plus}(X_2, 0, X_2) \leftarrow \text{nat}(X_2)$$

with MGU $\mu_0 = \left\{X_2 = s(s(s(0)), Z_1 = s(s(s(0)))\right\}$. The new goal is $G = \left\{\text{nat}(s(s(s(0)))\right\}$, the new MGU is obtained as

$$\mu' = \mu_0 \circ \mu = \left\{Z = s(s(s(s(0)))), X_2 = s(s(s(0)), Z_1 = s(s(s(0)))\right\},$$

from which we only keep the binding for Z:

$$\mu = \left\{Z = s(s(s(s(0))))\right\}.$$

Continuing this way will successively remove successor function symbols from our goal by means of the second clause, and finally remove the remaining nat(0) from G by means of the first clause. All this does not affect the binding for Z in our MGU μ anymore. The final μ can only contain this binding of the query variable, which establishes the answer to our query. In fact, $3 + 1 = 4$.

It is important to notice that the program can also by used for subtraction by means of queries like $\leftarrow \text{plus}(X, 2, 3)$. Moreover, a query $\leftarrow \text{plus}(X, Y, 3)$ would deliver a sample solution for the equation $X + Y = 3$. We thus see that the resolution algorithm has in fact a very logical flavor. The validity of the query is constructively proven by finding a suitable variable binding. Unfortunately, with every new query, basic things like “3 is a natural number” are proven over and over again. Moreover, these proofs are performed on a processor that can handle natural numbers and, in principle, simply add them. On the basis of this observation, there have been numbers added to Prolog and *built-in predicates* on numbers.

One example is the binary predicate “IS.” It is notated infix. Atoms could now be of the form Z IS $3 + 1$ or, more generally, Z IS $X + Y$, where X, Y, and

Z are used also elsewhere within the corresponding clause. There are two highly relevant restrictions with this mechanism. First, when arriving at such an atom, all expressions on the right hand side of “IS” must be bound to numbers; recall that there has been some MGU applied at that time. If this is not the case, then the interpreter will abort with a run-time error. If it is the case, however, the addition will be carried out. At this point, the second restriction becomes relevant. Success or failure of the atom is decided via unification of the result of the addition with the left hand side of “IS.” Hence, $\leftarrow Z$ IS $3+1$ will result in $Z = 4$, $\leftarrow 4$ IS $3+1$ will result in “yes,” $\leftarrow 3 + 1$ IS $3 + 1$ will result in “no,” since $3 + 1$ cannot be unified with 4. Using addition as subtraction by asking $\leftarrow 3$ IS $X + 2$ will result in a runtime error. All this, in particular the required binding, forces the programmer to reflect about the resolution process. This contradicts the original idea of separating logic and control.

This dilemma has been resolved with the step from logic programming to *constraint logic programming* (CLP) around the mid of the eighties. CLP combines logical programming languages with *constraint solvers*. Constraint solving was another established declarative programming paradigm that had come into existence already in the early sixties in connection with graphics systems such as the famous Sketchpad [Sut63]. Constraint solvers have always played a considerable role within interactive graphics systems. One famous example is Thinglab [Bor79] developed around the end of the seventies. In this sense, a *constraint solving problem* is given by a finite set of *constraints*. A constraint is a relational dependence between several numbers, variables, and certain functions on these numbers and variables. The type of numbers and the possible functions and relational dependences make up the *domain* of the constraint solver. One example are systems of linear equations. Another example are linear programming problems (here, the target function can easily be coded as another constraint). In general, “numbers” can be, of course, any objects. A *solution* of a constraint system is *one* binding of all involved variables such that all constraints are simultaneously satisfied. A constraint solver computes such a solution if possible. In particular, it checks this way for *feasibility*, i.e., the existence of a solution.

Within CLP, constraints may appear, besides regular atoms, within the bodies of program clauses and within queries. Constraint solvers are supposed to admit at least *equations* as valid constraints. Then the concept of syntactic equality, which was the basis for the use of unification, is replaced by the concept of equality over the corresponding domain. Consequently, it is handled by the constraint solver. The resolution procedure sketched above maintained besides the current goal G a cumulative MGU μ. This MGU contained at the end the variable binding establishing the answer to the initial query. CLP replaces the MGU by a *constraint storage*, i.e., a set C of constraints. This storage is enlarged during resolution in two ways:

1. Constraints in the query or in the body of clauses never enter the current goal G but the constraint storage C.

2. Unification is replaced by passing the corresponding equality constraints to the constraint solver. An atom successfully matches a head of a clause iff these constraints are feasible. In the positive case they enter C.

The constraint storage C will always contain *implicit descriptions*, in contrast to explicit binding within an MGU, of the possible values for all variables involved in the processes described above. The step of removing variable bindings from the MGU is replaced by a *projection*, which the constraint solver is supposed to provide. This projection is essentially some restricted existential quantifier elimination. If the goal G becomes empty at the end, then C contains an implicit description of possible values for the variables from the initial query.

The initial step towards this type of systems was Colmerauer's Prolog II introducing negated equality "$\neq$" and an extended unification that could handle infinite cyclic terms, also called rational trees. Its foundations were already presented in terms of constraint solving [Col86]. Around 1988, three constraint logic systems of high influence appeared independently: CHIP [DVS$^+$88], CLP(R) [JMSY92], and Prolog III [Col90]. CHIP includes constraint solvers for arithmetic over finite domains and finite Boolean algebra. It also handles rational linear constraints by an extended Simplex algorithm. There are several commercial products based on CHIP marketed. CLP(R) provides the first implementation of a clean declarative treatment for arithmetic expressions within a logical programming language. The constraint solver can handle real linear arithmetic using floating point numbers. The method is again an extended Simplex algorithm. CLP(R) is commercially marketed. Colmerauer's Prolog III handles, besides Boolean constraints and finite lists, linear rational arithmetic using exact numbers. The method is once more based on the Simplex algorithm. Prolog III is also a commercial product.

Our work is going to extend the classical framework of CLP discussed so far by admitting as constraints arbitrary *first-order* formulas. Over the reals we have as relations equality "$=$," negated equality "$\neq$," weak ordering "$\leqslant$" and "$\geqslant$," and strict ordering "$<$" and "$>$." Instead of restricting to sets of such constraints, which are considered conjunctive, the constraints can be combined by means of the usual Boolean operators "$\neg$," "$\wedge$," "$\vee$," "$\longrightarrow$," "$\longleftarrow$," "$\longleftrightarrow$." In addition, existential quantification "$\exists x$" and universal quantification "$\forall x$" can be applied with the quantified variable x ranging over domain elements.

Our description will focus very much on the reals because this is a most interesting application area. Also it will be very familiar to the reader, and we consider it well-suited for making our point. Besides the reals $\mathbb{R}$, our system can compute over the complex numbers $\mathbb{C}$ and over p-adic numbers $\mathbb{Q}_p$ for fixed primes p. We will also describe these other domains and give some examples.

Hong [Hon93] has described a CLP system RISC-CLP(REAL). It allows for real constraints of arbitrary degree. They are solved by either real quantifier elimination or by Gröbner basis methods. The approach has, however, the usual restriction of pure lists of constraints in contrast to arbitrary first-order formulas.

There is considerable research being done on *first-order constraints*. This so far mostly affects the treatment of finite domains, finite or infinite terms, approximate methods, and decidability considerations for very general domains. The connection between researchers developing first-order methods on one hand, and researchers seeking such methods for incorporation into their CLP framework still appears to be very weak [Col01].

In a pure CLP approach, one would fix a domain, and then all variables of a program would be considered variables over this domain. Then the constraint solver completely replaces unification also for non-constraint subgoals. A more general approach would admit several types of variables: variables over the domain of the solver, variables over other domains treated by other solvers, and general variables to be traditionally treated by unification. All these extensions do not at all interfere with the issues that we are going to discuss here. We will thus allow ourselves to restrict to the basic case that all variables are elements of the domain of our solver, and that this solver is the only one involved.

In Section 2, we give a summary of real quantifier elimination, existing methods, and corresponding implementations. Section 3 gives an overview of the implementation in our CLP(RL) system and of the underlying REDLOG system. In Section 4, we describe in detail how our CLP interpreter works. Moreover, we will make clear that the use of quantifier elimination as a constraint solver conceptually extends the framework of CLP in a considerable way. The main points are:

- constraints of arbitrary degree,
- exact real arithmetic for arbitrary degree,
- absolutely clean treatment of disjunction,
- quantified constraints.

In Section 5 we demonstrate our capabilities for the available non-real domains, which are at present complex numbers $\mathbb{C}$, and the linear theory of p-adic numbers $\mathbb{Q}_p$ for primes p. The sections 4 and 5 include application examples and computing times to give an idea of the power of our system. In Section 6 we provide an outlook on the continuation of this research project. We have arrived at the point where both our concept and the implemented system provide a suitable basis for introducing

- parametric constraints.

In Section 7 we finally summarize our results and evaluate our work.

2 Real Quantifier Elimination

2.1 A Formal Framework

In order to give a formal framework for real quantifier elimination, we introduce first-order logic on top of polynomial equations and inequalities. For this, we start with the language $\mathcal{L} = (0, 1, +, -, \cdot\,; \leqslant, \geqslant, <, >, \neq)$ of ordered rings, and expand it by constants for all rational numbers yielding $\mathcal{L}(\mathbb{Q})$.

Then in $\mathcal{L}(\mathbb{Q})$ every term can be equivalently represented by a multivariate polynomial $f(u, x)$ with rational coefficients, where $u = (u_1 \ldots, u_m)$ and $x = (x_1, \ldots, x_n)$. We call u *parameters* and we call x *main variables*.

Equations will be expressions of the form $f = 0$, inequalities are of the form $f \leqslant 0$, $f \geqslant 0$, $f < 0$, $f > 0$, or $f \neq 0$. Equations and inequalities are called *atomic formulas*.

Quantifier-free formulas are "true," "false," atomic formulas, and any combination between these by the logical operators "$\neg$," "$\wedge$," "$\vee$," "$\longrightarrow$," "$\longleftarrow$," "$\longleftrightarrow$."

A formula of the form $\exists x_1 \ldots \exists x_n \psi(u, x)$, where $\psi(u, x)$ is a quantifier-free formula, is called an *existential formula*. Similarly, *universal formulas* are of the form $\forall x_1 \ldots \forall x_n \psi(u, x)$. A *prenex first-order formula* has several alternating blocks of existential and universal quantifiers in front of a quantifier-free formula. General *first-order* formulas are obtained from "true," "false," and atomic formulas by arbitrarily mixing and repeating the application of Boolean operators and quantification. It is not hard to see, however, that every first-order formula is equivalent to a prenex formula.

We denote by $\mathrm{Th}(\mathbb{R})$ the theory of the real numbers over the language $\mathcal{L}(\mathbb{Q})$. This is the set of all $\mathcal{L}(\mathbb{Q})$-sentences that hold over the reals. The real *quantifier elimination problem* can be phrased as follows: Given a formula φ, find a quantifier-free formula φ' such that both φ and φ' are equivalent in the domain of the real numbers, formally

$$\mathrm{Th}(\mathbb{R}) \models \varphi' \longleftrightarrow \varphi.$$

A procedure computing such a φ' from φ is called a real *quantifier elimination procedure*.

Quantifier elimination for an existential formula $\varphi(u) \equiv \exists x_1 \ldots \exists x_n \psi(u, x)$ has a straightforward geometric interpretation: Let

$$M = \left\{\, (u, x) \in \mathbb{R}^{m+n} \;\middle|\; \psi(u, x) \,\right\},$$

and let $M' = \{ u \in \mathbb{R}^m \mid \varphi(u) \}$. Then M' is the projection of M along the coordinate axes of the existentially quantified variables x onto the parameter space. Quantifier elimination yields a quantifier-free description of this projection.

2.2 Implemented Methods

For a thorough survey of the three implemented methods for real quantifier elimination and their application range, we point the reader at [DSW98]. Here, we are going to summarize the characteristics of the available systems as far as this is interesting for our purpose.

The first method to mention is *cylindrical algebraic decomposition* (CAD) [Col75]. This is the oldest and most elaborate implemented real quantifier elimination method. It was developed by Collins and his students starting in 1974. During the last 10 years particularly Hong made very significant theoretical contributions that improved the performance of the method dramatically resulting in *partial* CAD [CH91]. Besides Hong, Brown has recently considerably contributed to the progress in CAD-based quantifier elimination. The time complexity of partial CAD is double exponential in the number of variables, where there is no difference to be made between main variables and parameters. Its application range is quite universal. The elimination problems should however not contain too many variables.

Partial CAD is implemented in the program QEPCAD, which is originally by Hong. Meanwhile there are versions of QEPCAD available from both Brown and Hong, which are to some extent being merged together from time to time. There is a new implementation of partial CAD under development within the REDLOG system [DS97a], which provides the platform for the project discussed here. We will discuss REDLOG in more detail in the following Section 3.

The second implemented method is quantifier elimination by *virtual substitution*. This method dates back to a theoretical paper by Weispfenning [Wei88]. During the last ten years considerable theoretical progress has been made to improve the method. The applicability of the method in the implemented form is restricted to formulas in which the quantified variables occur at most quadratically. Moreover, as quantifiers are successively eliminated, the elimination of one quantifier can increase the degree of other quantified variables. There are various heuristic methods built in for decreasing the degrees during elimination. One obvious example for such methods is polynomial factorization. The time complexity of the virtual substitution method is double exponential in the number of changes between "$\exists$" and "$\forall$" in a prenex input formula. With like quantifiers, it is only single exponential in the number of main variables. The number of parameters does not contribute to complexity in any relevant manner. The method has turned out highly useful for problems containing comparatively many parameters.

There are surprisingly many practical applications where the degree restrictions are satisfied.

After promising experimental implementations by the author starting in 1992, the method has been efficiently reimplemented within the REDLOG system by the author together with A. Dolzmann.

The third method is based on *parametric multivariate real root counting*. The basis for this method is a theorem on real root counting for multivariate polynomials, which is an extension of a univariate theorem by Hermite (1853). It was found independently by Becker and Wörmann [BW94] and Pedersen, Roy, and Szpirglas [PRS93]. This approach can be extended to obtain the exact number of roots under several side conditions on the signs of some other polynomials. For real quantifier elimination, this root counting has to be further extended to multivariate polynomials with parametric coefficients in such a way that it will remain correct for every real specialization of the parameters including specializations to zero. This task has been carried out by Weispfenning using comprehensive Gröbner bases [Wei98]. There is no result on the theoretical complexity of the overall procedure. The method is quite special-purpose. There are, however, examples [DSW98] where it performs superior to the two other methods. As a rule, it performs well on input formulas containing many equations.

The method has been implemented by Dolzmann within the package QERRC of the computer algebra system MAS. There is a reimplementation within REDLOG in progress currently.

3 The CLP(RL) System

The CLP(RL) system is implemented on top the computer logic system REDLOG. RL is an abbreviation for REDLOG; inside REDLOG all user-available functions are prefixed with "`rl`"; thus the name. REDLOG itself is implemented within the computer algebra system REDUCE. As REDUCE forms the user interface to CLP(RL) all facilities of both REDLOG and REDUCE, as well as the Lisp system underlying REDUCE are fully available to the CLP(RL) user. Moreover REDLOG provides interfaces to QEPCAD and QERRC. Figure 1 depicts the situation.

3.1 Redlog

REDLOG stands for "REDUCE logic" system. It provides an extension of the computer algebra system REDUCE to a computer logic system implementing symbolic algorithms on first-order formulas wrt. temporarily fixed first-order languages and theories. Underlying theories currently available are algebraically closed fields

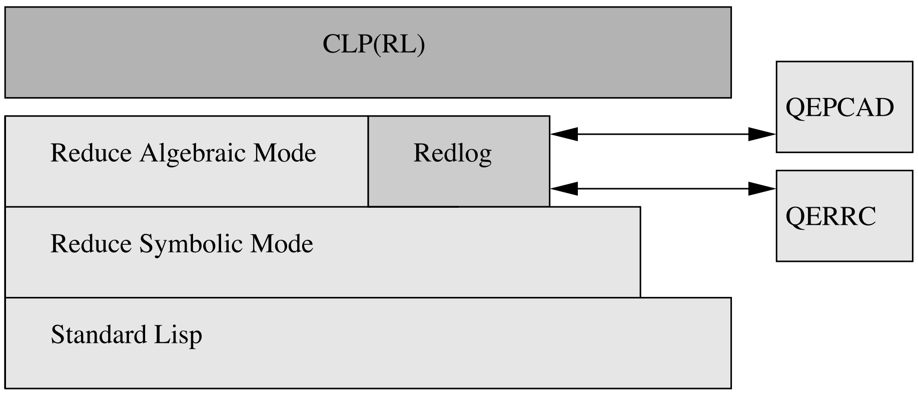

Figure 1: The layered design of CLP(RL)

(“complex numbers”), real closed fields (“real numbers with order”), and discretely valued fields (“*p*-adic numbers”).

REDLOG origins from the implementation of real quantifier elimination by virtual substitution. Successfully applying such methods to both academic and real-world problems, the authors have developed a large collection of formula-manipulating tools, many of which are meanwhile interesting in their own right.

Based on experimental implementations beginning in 1992, the author started the realization of REDLOG in early 1994. In April 1995, the system has been redesigned by the author together with A. Dolzmann [DS97a]. REDLOG 1.0 had been published on the Web in October 1996. This first REDLOG distribution was a great success. Meanwhile REDLOG has become a part of the REDUCE system. The current version REDLOG 2.0 is part of the the current REDUCE 3.7 of April 1999. REDLOG is widely accepted inside the scientific community. For a collection of applications of REDLOG in various fields see [DSW98] and the references there.

3.2 CLP(RL)

CLP(RL) provides support for inputting CLP programs in the algebraic mode of REDUCE. The notation for clauses follows the usual conventions: “$\longleftarrow$” is input as “`:-`”. Programs are lists of clauses. There is then a function `clp` expecting two arguments: a program and a query. The return values of `clp` is a quantifier-free formula in the query variables, which can then be further processed within the algebraic mode of REDUCE extended by REDLOG and CLP(RL).

The function `clp` implements the constraint-based resolution procedure described in the following section. For quantifier elimination it uses by default the REDLOG procedure of the currently selected language and theory. Optionally any version of QEPCAD or QERRC can be used.

4 The Resolution Algorithm

A *constraint* is a first-order formula over the language $\mathcal{L}(\mathbb{Q})$ as in Section 2.1. An *atom* is of the form $P(t_1, \dots, t_n)$ for $n \in \mathbb{N}$ where P is an n-ary predicate symbol that is not in $\mathcal{L}(\mathbb{Q})$, and the $t_1, \dots, t_n$ are terms.

A *clause* is of the form $\beta_0 \leftarrow \beta_1, \dots, \beta_n, \psi$, where $\beta_0, \dots, \beta_n$ are atoms, and ψ is a constraint. The atom β_0 is the *head* of the clause. The sequence $\beta_1, \dots, \beta_n, \psi$ is the *body* of the clause. Notice that it is not possible to have a constraint as the head of a clause. A *program* is a finite set of clauses. A *query* is of the form $\leftarrow \alpha_1, \dots, \alpha_n, \varphi$, where $\alpha_1, \dots, \alpha_n$ are atoms, and φ is a constraint.

Let Π be a program, and let Q be a query. Then we can fix an expansion language $\mathcal{L}(\mathbb{Q})' \supseteq \mathcal{L}(\mathbb{Q})$ containing all predicate symbols and function symbols occurring in Π and Q. Over this language $\mathcal{L}(\mathbb{Q})'$ we identify clauses

$$\beta_0 \leftarrow \beta_1, \dots, \beta_n, \psi$$

with first-oder formulas $\beta_0 \longleftarrow \beta_1 \land \dots \land \beta_n \land \psi$. Accordingly, we identify the program Π with the conjunction $\bigwedge \Pi$ of the contained clauses. Finally, the empty head in the query Q is interpreted as "false," and thus $Q = \leftarrow \alpha_1, \dots, \alpha_n, \varphi$ is identified with $\neg(\alpha_1 \land \dots \land \alpha_n \land \varphi)$. Recall that "true" is a constraint, and that forming conjunctions is a valid operation for constructing constraints. From this point of view, it is no restriction that clauses and queries contain exactly one constraint.

Let Π be a program. The *completion* $\bar{\Pi}$ of Π is obtained by adding to Π for each n-ary predicate symbol P in Π a first-order formula γ as follows: Let P be defined by clauses

$$P(t_{11}, \dots, t_{1n}) \leftarrow B_1, \quad \dots, \quad P(t_{m1}, \dots, t_{mn}) \leftarrow B_m.$$

Let $x_1, \dots, x_n$ be pairwise distinct variables not occurring in these clauses. Denote for $i \in \{1, \dots, m\}$ by $y_{i1}, \dots, y_{ik_i}$ the variables occurring in the i-th clause above. Then γ is given by

$$P(x_1, \dots, x_n) \longleftrightarrow \bigvee_{i=1}^{m} \exists y_{i1} \dots \exists y_{ik_i} \Big(\bigwedge_{j=1}^{n} x_i = t_{ij} \land B_i \Big).$$

Note that in the special case that P does not occur in the head of a clause, this amounts to $P(x_1, \dots, x_n) \longleftrightarrow$ false, which is equivalent to $\neg P(x_1, \dots, x_n)$.

A variable v is *free* in a first-order formula φ if v occurs in φ outside the scope of all quantifiers $\exists v$ and $\forall v$. We denote by $\operatorname{var}(\varphi)$ the set of all variables that are free in φ. Let $V = \{v_1, \dots, v_k\}$ be a finite set of variables. Then $\exists V \varphi$ is a concise notation for $\exists v_1 \dots \exists v_k \varphi$. Even more concisely, $\underline{\exists}\varphi$ stands for the *existential closure* $\exists \operatorname{var}(\varphi)\, \varphi$, and $\underline{\forall}\varphi$ stands for the *universal closure* $\forall \operatorname{var}(\varphi)\, \varphi$.

In view of our identification of syntactic entities with first-order formulas above, all these definitions can obviously be applied also to the former.

Let Π be a program, and let $Q = \leftarrow \alpha_1, \dots, \alpha_k, \varphi$ be a query. Then a *correct answer* for Π and Q is a quantifier-free constraint φ' such that $\operatorname{var}(\varphi') = \operatorname{var}(Q)$ and

$$\operatorname{Mod}\bigl(\bar{\Pi} \cup \operatorname{Th}(\mathbb{R})\bigr) \models \underline{\forall}(\varphi' \longrightarrow \alpha_1 \land \cdots \land \alpha_k \land \varphi).$$

Note that the model class is over the expanded language $\mathcal{L}(\mathbb{Q})'$ while $\operatorname{Th}(\mathbb{R})$ always denotes the $\mathcal{L}(\mathbb{Q})$-theory of the reals.

Besides quantifier elimination, our following resolution algorithm applies a *simplifier* to the final result formula. Simplifiers map first-order formulas to simpler equivalent ones. In the case of the reals, we use a simplification method based on Gröbner bases. For details and a discussion of the notion of simplicity see [DS97b].

Algorithm 1 *Input: A program* Π *and a query* $Q = \leftarrow \alpha_1, \dots, \alpha_k, \varphi$. *Output: A correct answer for* Π *and* Q. *Termination: The algorithm does not necessarily terminate. Used subroutines:* simplify *is a simplifier;* qe *is a quantifier elimination procedure.*

```
begin
    $(G', C') := \operatorname{clpqe}\bigl(\{\alpha_1, \dots, \alpha_k\}, \varphi\bigr)$
    $C' := \operatorname{simplify}(C')$
    return $C'$
end
procedure clpqe$(G, C)$
begin
    $V := \operatorname{var}(C) \setminus \bigl(\operatorname{var}(G) \cup \operatorname{var}(Q)\bigr)$
    $C := \operatorname{qe}(\exists V C)$
    if $G = \varnothing$ or $C = \mathrm{false}$ then
        return $(G, C)$
    fi
    while $G \neq \varnothing$ do
        remove $P(t_1, \dots, t_n) \in G$ from $G$
        standardize apart all variables in $\Pi$
        if exists $P(s_1, \dots, s_n) \leftarrow B, \psi \in \Pi$
            s.t. setting $\mu := \bigwedge_{i=1}^{n} s_i = t_i$ we have $\operatorname{qe}\bigl(\underline{\exists}(\mu \land C)\bigr) = \mathrm{true}$
        then
            $G := G \cup B$
            $C := C \land \mu \land \psi$
            return clpqe$(G, C)$
        else
```

```
            return $(G, \text{false})$
        fi
    od
    return clpqe$(\varnothing, C)$
end
```

Given the program $\Pi = \{p(X) \leftarrow X \geqslant 0, p(X) \leftarrow X \leqslant 0\}$ and the query $\leftarrow p(X)$, it is clear that "true" is a correct answer. There are, however, exactly two possible answers that can be computed by the algorithm: $X \geqslant 0$ and $X \leqslant 0$. The algorithm is complete only in the following sense:

> *Let* Π *be a program, let* Q *be a query, and let* φ' *be a correct answer for* Π *and* Q*. Then there are finitely many runs of Algorithm 1 on* Π *and* Q *with corresponding results* $\varphi'_1, \dots, \varphi'_r$ *such that* $\mathrm{Th}(\mathbb{R}) \models \underline{\forall}(\varphi' \longrightarrow \bigvee_{i=1}^{r} \varphi'_i)$.

A proof for this restricted completeness as well as for the correctness of Algorithm 1 can be derived from the corresponding proofs for other constraint solvers in any textbook on CLP. It is not hard to see that our quantifier eliminations exactly provide the various services required from constraint solvers there.

So far, we resolve the non-determinism in Algorithm 1 exclusively by selecting the first possible program clause in the order of notation and maintaining a stack of goals. Though common in logic programming, this is a considerable restriction, because it can lead to infinite runs of Algorithm 1 where there would exist finite runs yielding correct answers.

We are now going to discuss the most striking features of our extremely general and powerful approach.

4.1 Constraints of Arbitrary Degree

By default CLP(R) uses the extremely efficient quantifier elimination by virtual substitution in REDLOG. In spite of the degree restrictions mentioned in Section 2.1, we are not at all restricted to linear constraints even in this case. As an example, consider the computation of Pythagorean triples, i.e., natural numbers $x, y, z \in \mathbb{N}$ with $x^2 + y^2 = z^2$. We use the following program:

$$\begin{aligned} \mathrm{nat}(0) &\leftarrow \\ \mathrm{nat}(X+1) &\leftarrow \mathrm{nat}(X),\ X \geqslant 0 \\ \mathrm{pyth}(X, Y, Z) &\leftarrow \mathrm{nat}(X),\ \mathrm{nat}(Y),\ \mathrm{nat}(Z),\ 2 \leqslant X \leqslant Y \leqslant Z \land X^2 + Y^2 = Z^2 \end{aligned}$$

The query $\leftarrow \mathrm{pyth}(3, 4, Z)$ yields $Z - 5 = 0$ in 0.05 s. For $\leftarrow \mathrm{pyth}(X, 9, Z)$ we obtain $X - 12 = 0 \land Z - 15 = 0$ in 0.7 s, and $\leftarrow \mathrm{pyth}(X, Y, 9)$ results in "false" after 0.3 s. A completely parametric query $\leftarrow \mathrm{pyth}(X, Y, Z)$ would result in an infinite run.

Our next example is taken from Hong [Hon93]. The program describes the Wilkinson polynomial equation:

$$\mathrm{wilkinson}(X, E) \quad \leftarrow \quad \prod_{i=1}^{20}(X + i) + EX^{19} = 0$$

Mind that this product actually occurs in the program in the following expanded polynomial form:

$$\begin{aligned}
X^{20} &+ (210 + E)X^{19} + 20615X^{18} + 1256850X^{17} + 53327946X^{16} \\
&+ 1672280820X^{15} + 40171771630X^{14} + 756111184500X^{13} \\
&+ 11310276995381X^{12} + 135585182899530X^{11} + 1307535010540395X^{10} \\
&+ 10142299865511450X^{9} + 63030812099294896X^{8} \\
&+ 311333643161390640X^{7} + 1206647803780373360X^{6} \\
&+ 3599979517947607200X^{5} + 8037811822645051776X^{4} \\
&+ 12870931245150988800X^{3} + 13803759753640704000X^{2} \\
&+ 8752948036761600000X + 2432902008176640000.
\end{aligned}$$

On the query $\leftarrow \mathrm{wilkinson}(X, 0),\ -20 \leqslant X \leqslant -10$ we obtain after 0.3 s the answer

$$\bigvee_{i=1}^{20} X + i = 0.$$

For the query $\leftarrow \mathrm{wilkinson}(X, 2^{-23}),\ -20 \leqslant X \leqslant -10$ with a slight perturbation, we obtain after 0.9 s the following answer (in expanded form):

$$8388608 \cdot \Big(\prod_{i=1}^{20}(X + i) + 2^{-23}X^{19}\Big) = 0 \wedge X + 20 \geqslant 0 \wedge X + 10 \leqslant 0.$$

The integer factor is the least common denominator of the coefficients of the product polynomial. This answer is contradictory. This could be tested, e.g., by applying quantifier elimination to its existential closure. Hong's RISC-CLP(REAL) actually delivers the result "false." It generally applies some sophisticated processing to its results including DNF computation at the risk of exponentially increasing the size of the output. Since our CLP(RL) lives inside a computer algebra system, we prefer to leave the responsibility of how to proceed to the user. In this situation it would be straightforward to apply the partly numerical function `realroots` of

REDUCE to the left hand side polynomial of the equations. This yields after 0.5 s the result

$$X \in \{-20.8469, -8.91725, -8.00727, -6.9997, -6.00001, -5, -4, -3, -2, -1\}.$$

If one wishes to remain exact, one could, e.g., apply QEPCAD to the existential closure of the answer, which immediately yields "false." In general, numerical methods will be more efficient, of course.

4.2 Exact Arithmetic

The minimal perturbation of Wilson's equation in the previous section has dramatically demonstrated how sensitive the root behavior of polynomials and thus algebraic equations and inequalities are even to smallest rounding errors. Within CLP(RL) all arithmetic is exact. The price is that we possibly obtain only implicit solutions as we have also seen in the previous section. Then one has the choice to either remain exact, or to apply approximate methods.

As long as we are within the CLP framework, however, we remain absolutely exact, and the answers—though not necessarily explicit—are always of the best possible quality from the point of view of exactness.

4.3 Disjunction

Recall that in traditional CLP, constraints are finite sets of relational dependences, which are regarded as conjunctions. There has been a considerable discussion about disjunctions of constraints within clauses and corresponding modifications of the resolution algorithm for treating certain restricted variants of disjunction in an appropriate way. All suggested solutions eventually led to further restrictions of completeness such that one did not really obtain a procedural counterpart to the declarative meaning of disjunction.

Within our framework, disjunction is most naturally and absolutely completely handled by the constraint solver itself. Our resolution algorithm does not at all know about the possible existence of disjunctive constraints. One standard example when discussing disjunctive constraints is the minimum function. Our program for this would look as follows:

$$\min(X, Y, Z) \quad \leftarrow \quad (X \leqslant Y \land Z = X) \lor (Y \leqslant X \land Z = Y)$$

The answers that can be derived from this program are as complete and concise as the definition itself. For the query $\leftarrow \min(3, 4, Z)$ we obtain $Z - 3 = 0$. For $\leftarrow \min(X, Y, 3)$ the answer is

$$(X - 3 = 0 \land Y - 3 \geqslant 0) \lor (X - 3 \geqslant 0 \land Y - 3 = 0).$$

Asking for $\leftarrow \min(X, Y, Z)$, we obviously get the definition itself. These computations take no measurable time.

4.4 Quantified Constraints

Since our constraints are first-order formulas, they may also contain quantification. It follows, of course, from the existence of a quantifier elimination procedure for the reals that this does not increase the expressiveness. On the other hand, it supports the concise formulation of programs.

The following program, for instance, describes that in real 2-space the point (u_1, u_2) is the image of the point (x_1, x_2) under central projection from the punctual light source (c_1, c_2):

$$\mathrm{pr}(C_1, C_2, X_1, X_2, U_1, U_2) \quad \leftarrow \quad \exists T\Big(T > 0 \wedge \bigwedge_{i=1}^{2} U_i = T(X_i - C_i)\Big).$$

Notice that this description covers all degenerate cases that arise when some of the points or coordinates coincide. The following is a possible quantifier-free description with 10 atomic formulas:

$$\begin{aligned}
&(C_1 = 0 \wedge C_2 = 0 \wedge U_1 = X_1 \wedge U_2 = X_2) \vee \\
&(C_2 \neq 0 \wedge C_2U_2 > C_2X_2 \wedge C_1U_2 - C_1X_2 - C_2U_1 + C_2X_1 = 0) \vee \\
&(C_1 \neq 0 \wedge C_1U_1 > C_1X_1 \wedge C_1U_2 - C_1X_2 - C_2U_1 + C_2X_1 = 0).
\end{aligned}$$

The quantified formulation is taken from [SW98], the quantifier-free result is obtained by quantifier-elimination with REDLOG. In higher dimension the effect becomes more dramatic. The quantifier-free description in 3-space has 18 atomic formulas, the one in 4-space 28.

5 Beyond Real Numbers

The design of CLP(RL) exclusively depends on the existence of a quantifier elimination procedure over the considered domain. Although the theory of real closed fields is the by far most prominent example for effective quantifier elimination, such procedures exist also for other theories. REDLOG itself implements quantifier elimination also for algebraically closed fields, i.e., complex numbers, and for *p*-adically closed fields, i.e., *p*-adic numbers for primes *p* [Stu00]. The *p*-adic quantifier elimination is restricted to linear formulas. Both elimination procedures can be used within CLP(RL) without any restrictions.

The language used for the p-adic theory is kept one-sorted by coding ordering between values as abstract divisibilities: Let $x, y \in \mathbb{Q}_p$, then we define

$$x \mid y :\longleftrightarrow v(x) \leqslant v(y), \quad x \parallel y :\longleftrightarrow v(x) < v(y).$$

The following program is analogous to the definition of nat in Section 4.1. It defines the powers of the prime p. For each query, p must be chosen to be a fixed prime.

$$\begin{aligned} \mathrm{ppow}(1) &\leftarrow \\ \mathrm{ppow}(p \cdot X) &\leftarrow \mathrm{ppow}(X),\ 1 \mid X \end{aligned}$$

The constraint $1 \mid X$ states that X is a p-adic integer. It is obvious that successive division by p eventually leads to a number with negative value. Thus the constraint can play the role of the emergency brake such as $X \geqslant 0$ does in the definition of nat. For $p = 101$ the query

$$\leftarrow \mathrm{ppow}(1220190039947966824482749091552564 1902001)$$

yields "true" after 0.1 s. If we increase this number, which is 101^{20}, by 1, then the corresponding query immediately yields "false." In this case, the constraint solver recognizes that $\frac{101^{20}+1}{101}$ is not a p-adic integer.

6 Parametric Constraints

The next conceptual extension to what we have described so far is support for *parametric constraints*. A variable that is free in a constraint is considered a *parameter* if it does not occur elsewhere in the clause or query in which the constraint is located. Parameters are not standardized apart. The existential closure in the qe call replacing unification also applies to parameters. They are, however, never projected away.

After a successful run, the answer contains variables from the query and parameters. When fixing the parameters in the answer to domain elements, the answer becomes equivalent to the one that would have been obtained when fixing these parameters in the program and in the query before the run. REDLOG provides rich facilities for the further processing and evaluation of such parametric answers.

7 Conclusions

We have introduced an extension of CLP, where the constraints are arbitrary first-order formulas over some domain. Constraint solving consists in various applications of quantifier elimination. Our approach is implemented in our system

CLP(RL) based on REDLOG. Implemented domains are at the moment $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{Q}_p$. The advantages of our approach include constraints of arbitrary degree, exact arithmetic, absolutely clean treatment of disjunction and other Boolean operators, and quantified constraints. Our approach delivers a basis for considering parametric constraint solving as a next step.

References

[Bor79] Alan Hamilton Borning. *Thinglab—A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, July 1979.

[BW94] Eberhard Becker and Thorsten Wörmann. On the trace formula for quadratic forms. In William B. Jacob, Tsit-Yuen Lam, and Robert O. Robson, editors, *Recent Advances in Real Algebraic Geometry and Quadratic Forms*, volume 155 of *Contemporary Mathematics*, pages 271–291. American Mathematical Society, Providence, RI, 1994.

[CH91] George E. Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, September 1991.

[Col75] George E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages. 2nd GI Conference*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, Berlin Heidelberg, 1975.

[Col86] Alain Colmerauer. Theoretical model of Prolog II. In Michel van Canegham and David H. D. Warren, editors, *Logic Programming and its Applications*, pages 3–31. Ablex Publishing Corporation, Norwood, NJ, 1986.

[Col90] Alain Colmerauer. Prolog III. *Communications of the ACM*, 33(7):70–90, July 1990.

[Col01] Alain Colmerauer, editor. *Workshop on First-Order Constraints. Abstracts of talks*, Marseille, France, May 2001. Available as `http://alain.colmerauer.free.fr/resumes.pdf`.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of Prolog. *SIGPLAN Notices*, 28(3):37–52, March 1993.

[DS97a] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

[DS97b] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, August 1997.

[DSW98] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. Real quantifier elimination in practice. In B. H. Matzat, G.-M. Greuel, and G. Hiss, editors, *Algorithmic Algebra and Number Theory*, pages 221–247. Springer, Berlin, 1998.

[DVS$^+$88] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, Decemeber 1988*, pages 693–702. Ohmsha Publishers, Tokyo, 1988.

[Hon93] Hoon Hong. RISC-CLP(Real): Constraint logic programming over real numbers. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Kow79] Robert A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–435, July 1979.

[PRS93] Paul Pedersen, Marie-Françoise Roy, and Aviva Szpirglas. Counting real zeroes in the multivariate case. In F. Eysette and A. Galigo, editors, *Computational Algebraic Geometry*, volume 109 of *Progress in Mathematics*, pages 203–224. Birkhäuser, Boston, Basel; Berlin, 1993.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, January 1965.

[Stu00] Thomas Sturm. Linear problems in valued fields. *Journal of Symbolic Computation*, 30(2):207–219, August 2000.

[Sut63] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 23, pages 329–346, Detroit, Michigan, May 1963.

[SW98] Thomas Sturm and Volker Weispfenning. Computational geometry problems in Redlog. In Dongming Wang, editor, *Automated Deduction in Geometry*, volume 1360 of *Lecture Notes in Artificial Intelligence (Subseries of LNCS)*, pages 58–86. Springer, Berlin Heidelberg, 1998.

[War77] David H. D. Warren. *Implementing Prolog—Compiling predicate logic programs*. PhD thesis, University of Edinburgh, 1977.

[Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1&2):3–27, February–April 1988.

[Wei98] Volker Weispfenning. A new approach to quantifier elimination for real algebra. In B.F. Caviness and J.R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 376–392. Springer, Wien, 1998.