1 The logic programming paradigm in numerical computation

M.H. van Emden*

University of Victoria

Abstract. Although CLP(**R**) is a promising application of the logic programming paradigm to numerical computation, it has not addressed what has long been known as "the pitfalls of [numerical] computation" [12]. These show that rounding errors induce a severe correctness problem wherever floating-point computation is used. Independently of logic programming, constraint processing has been applied to problems in terms of real-valued variables. By using the techniques of interval arithmetic, constraint processing can be regarded as a computer-generated proof that a certain real-valued solution lies in a narrow interval. In this paper we propose a method for interfacing this technique with CLP(**R**). This is done via a real-valued analogy of Apt's proof-theoretic framework for constraint processing.

1.1 Introduction

In the first flowering of the logic programming paradigm, a large part of computer science was identified as suitable territory for conquest. This ambitious program suffered a significant omission: numerical computation. In this paper we will argue that there are correctness problems of great practical importance in numerical computation and that logic programming is a promising method to solve these. In the rest of this introduction we will trace in historical order the various steps needed to go from conventional numerical computation to a logic programming system for numerical computation.

Via interval arithmetic to sound numerics Correctness of numerical computation has not been fully addressed, either by the program verification community, or by logic programming. As we shall show below, this issue has also been ignored by mainstream numerical analysis. Only a distinct subculture, interval arithmetic, has taken seriously the possibility that users of numerical software might want guaranteed correct output.

In interval arithmetic one associates a set of reals with each real-valued unknown. Results of computations are given as membership of a set. This method is of practical value for two reasons:

* Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., V8W 3P6 Canada. Phone: (604) 721-7225, fax: (604) 721-7292, e-mail: vanemden@csr.uvic.ca. Home Page URL: http://www-csc.uvic.ca/home/vanemden/vanemden.html

1. The sets are restricted to closed, connected sets of reals such that its bounds, if any, are floating-point numbers. Such sets are called *intervals*. They are economically representable in computer memory, compared to other set representations.
2. Arithmetic is performed on intervals. As the result intervals can be computed by operations of a standard floating-point processor in such a way that correctness is preserved, the set operations are fast compared to most other set operations.

In this way numerical computations on standard floating-point processors can be interpreted as computer-generated proofs that a real-valued variable belongs to an interval that has a width near the limit imposed by the hardware. This method of ensuring correctness by combining intervals with modern floating-point hardware we call *sound numerics*.

Some proponents of interval arithmetic may not agree with the above redescription: other motivations and interpretations of interval arithmetic exist. However, its seems to us that sound numerics is the logical conclusion of a long development that started in the 1960s, even before Moore's landmark book [21].

Subsequent development of interval arithmetic Upon its inception in the 1960s, interval arithmetic was warmly received in the numerical computation community. There are, however, several reasons why the initial enthusiasm turned into disparagement.

1. Dependencies between variables cause intervals to become disappointingly wide. For example, if the interval associated with x is $[0, 1]$, then the one associated with $x - x$ is $[-1, 1]$.
2. Result intervals were not always correct. To guard against improper rounding, a fudge factor was subtracted from computed lower bounds and added to computed upper bounds. This slowed down computation and was difficult to do correctly in all cases without undue loss of accuracy.
3. In the prevailing culture of computing, higher quality of results has only been acceptable in combination with higher performance. We still have to get used to the idea that higher quality may come at a cost.

As a result of these conditions, interval methods were rejected by mainstream numerical computation. Only after this early rejection some of the potential of interval arithmetic was realized.

As to point 1 above, the disappointingly large intervals turned out to be a non-issue. These are problematic in the restricted context of evaluation of expressions. But expression evaluation is a minor concern in numerical computation. An example of a major one is the solving of a nonlinear equation. Here interval arithmetic, in spite of the dependency problem, makes possible the Interval Newton method. This method is not merely a sound alternative to what is achieved more efficiently by conventional numerical computation. Interval Newton proves that no solutions exists outside certain narrow intervals; it also may prove that such an interval contains exactly one or at least one solution. Such results have not been obtained by conventional numerical computation.

As for point 2 above, the IEEE standard for floating-point arithmetic has made superfluous the use of fudge factors. Instead, one can efficiently control the rounding modes to ensure that result intervals contain all values they should contain with a minimum of information loss. However, the last details of its use in interval arithmetic are still an area of active research [16].

As for point 3 above, the rule that the fast drives out the good has rarely been violated. Only in niche applications such as Lisp and Prolog, has security and programming convenience been considered an adequate reward for loss of execution speed. The recent wide-spread acceptance of other interpreters such as those for Perl and Java may be a sign that Gresham's law is relaxing its grip on computing.

Thus there are three reasons for believing that interval methods are experiencing a reversal of misfortune. The advent of interval constraints is yet another one.

Interval constraints Independently, Davis [9] and Cleary [7] arrived at a relational generalization of interval arithmetic, which is now known as *interval constraints*. Shortly after, Cleary's work was used in BNR Prolog [5,4] to obtain software that can be viewed in two different ways.

1. BNR Prolog as a version of Prolog where soundness is preserved for queries involving real numbers.
2. BNR Prolog as interval constraint system that happens to have Prolog as programming language front end.

Neither of these interpretations quite fits the logic programming paradigm: it is not clear how the decimals in the answer substitution make the answer a logical implication of the query. A step in this direction was taken by the CLP scheme.

CLP(**R**) The CLP scheme extends the scope of the logic programming paradigm in two ways. Semantically, it is an interface with theories that are important in applications, such as those for the integers, the reals, the regular expressions, and others. Operationally, it is an interface with important algorithms, such as Gaussian Elimination and Simplex.

Ironically, existing implementations of CLP(**R**) rely on conventional numerical computation. In this way the logic programming paradigm is compromised by the well-known pitfalls of floating-point computation. Forsythe [12] gave an early survey of the various ways in which the rounding errors of floating-point arithmetic can lead to nonsense output.

As illustration we restrict ourselves here to a particularly short and eloquent example, taken from Parker's paper [22]. Consider the recurrence relation

$$u_{k+1} = 111 - 1130/u_k + 3000/(u_k u_{k-1})$$

with initial values

$$u_0 = 2, u_1 = -4.$$

Computed results for u_{30}:

```
single precision  100.0000
double precision  100.00000000000
```

Parker reports that the value of u_{30} is between six and seven. As CLP(**R**) takes the results of floating-point computation at face value, one can "prove" anything by including computations such as these in the deduction.

What CLP(**R**) lacks are exactly the techniques of sound numerics.

Overview of this paper Errors in numerical computation can arise from a variety of sources, most of which are of no concern to numerical analysis, as conventionally practiced. In the logic programming paradigm one is responsible for the entire span between program specification and computer output.

If numerical computation is to be included in the logic programming paradigm, then numerical problems need to be specified in logic. This seems virgin territory, even for a problem as simple as the one addressed in Section 1.2: solving a single equation with a real unknown.

In the remaining sections we briefly recapitulate CLP(**R**) and propose a proof-theoretic approach to providing a sound completion of this method.

1.2 Numerical programs need verification

It is the aim of this paper to use logic programming to verify results obtained with numerical software. Before invoking the proposed tool, logic programming, let us consider what is required of *any* method of verification, whether logic programming or something else. The least one needs is a precise method for specifying the problem to be solved. Surprisingly, this turns out to be virgin territory. In this paper we consider the problem of formalizing a simple numerical problem: that of solving a single equation in a single unknown.

1.2.1 Specification of equation-solving

In single-variable equation solving, the problem is to find a real x such that $f(x) = 0$, where f is a real-valued function of a real-valued argument.

The need to specify it in formal logic brings forward the question: "What does it mean to solve $f(x) = 0$?" Solutions in the mathematical sense are reals that typically have no finite representation in any conventional number system. The shortest description of such a real may well be: "The least (or the second largest, or whatever) x such that $f(x) = 0$". That shows that we do not necessarily want a *solution* of $f(x) = 0$. We just want *useful information* about the reals being defined by $f(x) = 0$. Preferably information that is finite in quantity, such as a solution being between two floating-point numbers that are sufficiently close together.

Given that it is the ambition of logic programming to cover the entire gap between specification and computer output, let us ask again: What does it mean to solve $f(x) = 0$? What can we expect as output? To start with, for a typical f, $f(\alpha) \neq 0$ for all double-length IEEE-standard-standard floating-point numbers α. Suppose that, untypically, there exists a floating-point number α such that $f(\alpha) = 0$. Then usually $f^{FP}(\alpha) \neq 0$, where f^{FP} is the algorithm's floating-point implementation of f. And if, peradventure, $f^{FP}(\alpha) = 0$, then it is likely that $f(\alpha) \neq 0$.

The Numerical Zero Observations such as these will cause a numerical analyst to point out that it is an unspoken assumption that the purpose of computer software can at most be to find a *numerical* zero, which is any x such that $|f(x)| \leq \epsilon$ for a suitable $\epsilon > 0$.

However the idea of the Numerical Zero is only the first step. If one takes it too literally, one also runs into trouble. An algorithm for a Numerical Zero may be correct from the point of view of numerical analysis and still return an α such that $|f(\alpha)|$ is greater than ϵ.

After all, the algorithm finds that $|f^{FP}(\alpha)| \leq \epsilon'$, which is not inconsistent with $|f(\alpha)| > \epsilon$. There is usually a difference between $f(\alpha)$ and $f^{FP}(\alpha)$. The difference can be large if f^{FP} is ill-conditioned. Moreover, when ϵ is a nonzero power of ten, which happens a lot, then it is not equal to ϵ', which is a real with a finite binary representation. To make a long story short, there are plenty of possibilities for missing even such a realistic-sounding goal as the Numerical Zero.

Apart from these practical problems, there is a conceptual one. The obvious formalization of the Numerical Zero is $\exists x \in \mathcal{R}.\ |f(x)| \leq \epsilon$. This is subject to the same difficulty as before: The truth value of $\exists x \in \mathcal{R}.\ |f(x)| \leq \epsilon$ provides no information about the real that is asserted to exist. Such information is outside of the control of this formal specification and thus is prey to the well-known pitfalls of numerical computation.

Definition versus description Thus we see that "solving $f(x) = 0$" is ambiguous. It could be interpreted as deciding whether $\{x \in \mathcal{R} \mid f(x) = 0\}$ is empty. This concerns the *definition* of a zero. It could also mean, in addition, providing useful information about such reals as may satisfy the definition. This is a matter of finding a useful *description* of the real number being defined.

The conceptual difficulties arising in formal specification of solving $f(x) = 0$ are removed by distinguishing between definition and description. The definition is involved in the problem *statement*. The description is the required *result*.

Logic is suited equally well for formal definitions as for formal descriptions. A formula F with a free variable x can serve either as a description or as a definition: any object substituted for x that makes F true, satisfies the definition or description.

We formalize the distinction between definition and description as follows. Suppose we have formulas A and B, both with free variable x, that represent a definition and description respectively. Formula $\forall x.(A \Rightarrow B)$ can be interpreted as saying that what B describes is defined by A. The implication is necessary because it is often the case that we cannot show conclusively by numerical methods that a solution exists. Thus we cannot expect always to be able to show that A is true (definition is satisfied) *and* B is true (this description applies). In general the best we can hope for is a proof that, *if* there is a solution (A is true), then it is described by B.

To be of practical interest, A and B need to have a certain form.

1. They have to have the same free variables, which we will denote by the n-tuple x.
2. A has to be what we call a *Numerical Definition*: a conjunction of atomic formulas built up out of the vocabulary of a theory of real numbers. That is, each atom is an equality or inequality between terms denoting reals. This restricted form facilitates implementation, yet is expressive enough for a wide variety of numerical problems.
3. B has to be what we call an *Interval Description*. We assume that what one wants to know about real numbers is how they relate to other real numbers that we can represent on a computer; in other words, floating-point numbers. Taking into account that a solution in general is a tuple of reals and that there is in general more than one solution, it is reasonable to make an interval description a disjunction of atomic formulas of the form $x \in b$ where x is a tuple of n variables and b is the Cartesian product of n floating-point intervals. This restricted form facilitates implementation, yet is expressive enough for a wide variety of numerical problems.

Thus, the general form is

$$\forall x \in \mathcal{R}^n.((A_1 \wedge \cdots \wedge A_k) \Rightarrow (x \in b_1 \vee \cdots \vee x \in b_n)).$$

We call this a *Numerical Definition/Interval Description sentence*, and abbreviate it to *ND/ID sentence*.

Not all descriptions are useful. They are the more useful the smaller n and the smaller, for given n, $b_1, \ldots, b_n$ are. An important case of n being as small as possible is $n = 0$. In that case the sentence states unconditionally that there are no solutions.

The conditionality of the ND/ID sentence seems disappointingly weak. If there is no solution, then the sentence can be true with *any* ID formula. One should not lose sight of the fact that the ND/ID sentence states unconditionally that no solutions exist outside of the area designated by the ID formula.

Let us go back to the example of solving a single equation in a single unknown. In that case the ND/ID formula has the form

$$\forall x.(f(x) = 0 \Rightarrow (x \in [\alpha_1, \beta_1] \vee \cdots \vee x \in [\alpha_n, \beta_n])).$$

This formalization is suited to the limitations of numerical computation. If the solutions of $f(x) = 0$ are simple and well separated, then we can expect n to be equal to the number of zeroes and we can expect the intervals to be narrow. In pathological cases, any of the intervals can fail to contain a solution. This possibility cannot always be avoided because f may come so close to zero that the limited precision of the computer's arithmetic may fail to distinguish such a value of f from zero.

The constraint processing methods we consider in this paper have the property that at the location of such a false zero, the value of f is close to zero.

1.3 From Prolog to CLP(R)

One of the motivations of logic programming is that it yields results that are correct in the sense of being logically implied by the background knowledge. The role of the problem statement is to select that part of background knowledge that solves the problem. To play this role, the problem statement has to be expressed in logic.

Now that we have decided on a logic expression for numerical problems, let us see whether these can be solved in the logic programming paradigm. We start by tracing the development from Prolog [8], the first logic programming language.

Prolog has a subset that almost corresponds to a subset of first-order predicate logic in clausal form. The qualification is that most Prolog implementations omit the occurrence check in unification.

Exploiting the pure subset of Prolog was initially not a high priority among language designers and implementers. That defect has since been remedied by the language Gödel [17].

Pure Prolog realizes the following soundness result.

> If program P with query Q leads to success with substitution θ, then P logically implies $\forall x.(Q\theta)$, where x is the tuple of free variables in Q.

But the restrictions of pure Prolog are too severe to be of practical interest: data are restricted to trees based ultimately on symbolic constants.

1.3.1 Constraint logic programming

That we only get symbolic computation in Prolog is not surprising because the only computational step is goal reduction. There is no scope for software and hardware that implement any of the many powerful combinatorial and numerical algorithms that have been developed outside logic programming. The steps taken to overcome this limitation can be summarized under the term *constraint logic programming.*

In logic programming, all atoms in a condition are subject to goal reduction. In constraint logic programming an atom in a condition can be either a goal, to be treated by goal reduction as in logic programming, or a constraint. Constraints are not eliminated in the course of program execution. The conjunction of constraints is tested for consistency by extra-logical algorithms. A query, as generated during program execution, fails as soon as the conjunction of constraints is found to be inconsistent.

As in logic programming, execution terminates with success as soon as no goals remain in the query. In constraint logic programming there is typically in such a situation a nonempty conjunction of constraints. The answer is then conditional upon this residual conjunction. This describes the pioneering implementations of constraint logic programming: Prolog II and III.

The idea was formalized in the CLP scheme.

1.3.2 The CLP scheme

The CLP scheme is based on the observation that in logic programming the Herbrand base can be replaced by any of many other semantic domains. For example, the reals. In this section we first review the CLP scheme as described in [18].

As the CLP scheme can be used with different semantic domains, it has as parameter a tuple $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$ describing the semantic domain, where Σ is a signature, $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ-formulas, and $\mathcal{T}$ is a first-order Σ-theory. These components play the following roles. Σ determines the relations and functions that can occur in constraints. $\mathcal{D}$ is the structure over which computations are performed (for example the ordered field of the real numbers)[1]. $\mathcal{L}$ is the class of constraints that can be expressed. Finally, $\mathcal{T}$ axiomatizes properties of $\mathcal{D}$.

If a goal G has a successful derivation from program $\mathcal{P}$ with answer constraint C, then $\mathcal{P}, \mathcal{T} \models \forall[G \Leftarrow C]$. No substitution is applied to G because

[1] $\mathcal{D}$ is a structure (in the sense of model theory) consisting of a set D of values (the *carrier* of the structure) together with relations and functions over D as specified by the signature Σ. For example, the complete ordered field $\mathcal{R}$ has R, the set of real numbers, as carrier. The signature component of $\mathcal{R}$ specifies $\leq$ as relation, and $0, 1, +, -, \times, /$ as function symbols. The status of $=$ and $\neq$ varies between treatments: some include them in the signature; some regard them as part of logic.

in constraint logic programming unification is done by means of equations, which are part of the constraints.

Derivations in the CLP scheme are defined by means of *transitions* between states. A state is defined as a tuple $\langle A, C, S\rangle$ where A is a multiset of atoms and constraints and C and S are multisets of constraints[2]. Together C and S are called the *constraint store*. The constraints in C are called the *active constraints*; those in S the *passive constraints*.

The query Q corresponds to the initial state $\langle Q, \emptyset, \emptyset\rangle$. A successful derivation ends in a state of the form $\langle \emptyset, C, S\rangle$. The existence of such a derivation implies that

$$\mathcal{P}, \mathcal{T} \models \forall[(Q \Leftarrow (C \wedge S))]$$

.

The CLP scheme provides a framework for constraint store management by defining a derivation as a sequence of states such that each next state is obtained from the previous one by a transition.

There are four transitions:

1: Resolution

$$\langle A \cup \{a\}, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup \{s_1 = t_1, \ldots, s_n = t_n\}\rangle$$

if a is the atom selected out of $A \cup \{a\}$ by the computation rule, $h \leftarrow B$ is a rule of $\mathcal{P}$, renamed to new variables, and if $h = p(t_1, \ldots, t_n)$ and $a = p(s_1, \ldots, s_n)$.

$$\langle A \cup \{a\}, C, S\rangle \rightarrow_r \ \textit{fail}$$

if a is the atom selected by the computation rule, and for every rule $h \leftarrow B$ in $\mathcal{P}$, h and a have different predicate symbols.

2: Constraint Transfer

$$\langle A \cup \{c\}, C, S\rangle \rightarrow_c \langle A, C, S \cup \{c\}\rangle$$

if constraint c is selected by the computation rule.

3: Constraint Store Management

$$\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$$

if $\langle C', S'\rangle$ is inferred from $\langle C, S\rangle$.

4: Consistency Test

$$\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$$

if C is consistent.

$$\langle A, C, S\rangle \rightarrow_s \ \textit{fail}$$

if C is inconsistent.

[2] We will often regard C and S as formulas. Then they are the conjunctions of the constraints they contain as multisets.

1.3.3 CLP(R)

How does CLP(**R**) fit into the CLP scheme? In the first place, the semantic domain that is a parameter in the CLP scheme is instantiated to a theory of the reals. In addition, the CLP scheme is customized by a constraint store management strategy. The active constraints are linear equations or inequalities. They are solved by, respectively, Gaussian elimination and Simplex. Hopefully, this results in additional variables being instantiated, so that some passive constraints become linear, so that they can be added to the active constraints. The aim is to remove all passive constraints this way.

Existing implementations of CLP(**R**) implement the algorithms to solve the generated equations in the same way as in conventional numerical analysis. As a result, rounding errors prevent CLP(**R**) answers from being logical consequences. Thus such implementations should not be considered logic programming systems, but numerical problem solvers to be compared with the like of MatLab. Such a comparison will give CLP(**R**) advantages such as the ability to handle certain nonlinear problems via its unique linearization scheme.

1.4 Sound CLP(R)

There are two methods for obtaining soundness with floating-point computation: interval arithmetic and interval constraints. The most straightforward way of protecting the soundness of CLP(**R**) is to implement its algorithms for Gaussian elimination and Simplex method in interval arithmetic. But it should be kept in mind that, in doing so, one is up against all the obstacles that have prevented, for three decades, a wide adoption of interval arithmetic. The experience has been that translating an algorithm that works well in conventional computation gives disappointingly large intervals if converted unchanged to interval arithmetic.

For this reason it is attractive achieve sound CLP(**R**) by means of interval constraints rather than interval arithmetic[3], as was proposed in [26]. In the present paper we supply some of the details that are missing in [26]. In supplying these details it turned out that the distinction in the CLP scheme between active and passive constraints is not useful. Thus we consider a simplified version of the scheme where there is only one type of constraint. We will not spell out in detail the transitions of the CLP scheme described in section 1.3.2 according to this simplification. Suffice it to say that the state $\langle A, C, S\rangle$ becomes $\langle A, C\rangle$ and that the constraint-store management transition is dropped.

[3] Why are large intervals no problem in interval constraints, which is otherwise so closely related to interval arithmetic? The explanation is not simple. But it is easy to convince oneself that it is so by considering the performance of Numerica [15] where superior time performance is obtained as well as very small intervals.

In CLP($\mathbf{R}$), the leaf nodes of the search tree for query G are distinguished by having an empty goal conjunction. Thus a leaf node contains only a constraint conjunction, say, C. Such a node represents the conditional answer $\forall x.(G\theta \Leftarrow C)$ (see footnote[4]). C is a conjunction of equalities or inequalities between terms denoting reals. This is general enough to cover many important problems of numerical computation.

Let us consider the case where C is a conjunction expressing an equation that would be written as $f(x) = 0$ in a conventional informal discussion. Do we just want to know whether the equation has a solution? If so, that information would allow us to improve the conditional answer $\forall x.(G\theta \Leftarrow C)$ to $\exists x.G\theta$. For example,

$$\forall x.(p(g(x)) \Leftarrow ((x+1)(x-1) = 0))$$

would be "improved" to $\exists x.p(g(x))$ without a hint as to what such an x might be. It is reasonable to expect some useful information about the x that exists.

If this problem sounds familiar, it is because in section 1.2.1 we encountered the same puzzle when we were considering logic specification of equation-solving — at that time independently of the CLP scheme. In that section we concluded the need to distinguish between definition and description. We found that ND/ID sentences express both aspects. We saw that the conditionality of these formulas cannot always be avoided, so that they are useful as a generally applicable method for logically specifying numerical problems.

It is then clear what to do: to compute for every leaf node with constraint conjunction C an ND/ID formula where the ND part is C and where the ID part is as informative as we can make it. This desideratum often results in the ID part being the empty disjunction, i.e. the logical constant FALSE. In such a case $\neg\exists x.C$ has been proved and the leaf node can be omitted from the search tree. In case the ID part is not equivalent to FALSE, it is still possible that $\neg\exists x.C$. However, as we explain below, the method of interval constraints can make this combination of outcomes unlikely. Even then we still have valuable information: the contrapositive of the ND/ID formula guarantees that no solution to C exists outside the area described by the ID part.

[4] In the CLP scheme there are no explicit substitutions. Instead, equations are added to the constraint store; see the resolution transition in the description of CLP derivations. Although this has an attractive elegance, it has its advantages to be able to say that the constraint store specifies only the numerical problem to be solved. That is why we assume that substitutions are used in the same way as in the conventional way of describing SLD derivations [19].

1.5 Proving ND/ID formulas

For a given constraint conjunction C we need to prove an ND/ID formula

$$\forall x.(C \Rightarrow (x \in b_1 \vee \cdots \vee x \in b_k))$$

where the ID part is as informative as possible.

We achieve this goal by means of interval constraints in a proof-theoretic framework inspired by that given by Apt [1] for constraints over discrete domains. Just as Apt's method is a proof-theoretic model for constraint processing over integers as pioneered in CHIP [10], so also his method can be used as a proof-theoretic model for interval constraints over the reals. We adapt Apt's inference rules to domains of reals and show how they can be used to prove useful ND/ID formulas.

1.5.1 CSPs

Apt's proof-theoretic framework defines Constraint Satisfaction Problems (CSPs) and formulates proof rules that derive from a given CSP one or more other ones. In our development of CSPs, we start with the part that is generally applicable. After that we continue with CSPRs, that is, CSPs intended for constraints over reals.

A CSP may be described as follows.

- Syntactically, a CSP is an expression of the form $C\diamondsuit D$ where C is a conjunction of atomic formulas having a tuple $x = (x_1, \ldots, x_n)$ of free variables. D is called the domain expression and has the form $x \in b$ where b is the Cartesian product of the domains of $x_1, \ldots, x_n$.
- The semantics of CSPs is determined by assigning to $C\diamondsuit D$ as meaning the first-order predicate logic sentence $\exists x.(C \wedge D)$. In any particular use of CSPs we assume an application-specific theory, for example a theory for the reals and for floating-point intervals. A CSP is *solvable* if its meaning is logically implied by the assumed application-specific theory.
- A CSP $C\diamondsuit(x \in b)$ is *failed* if b is empty.
- A *solution* of a CSP is a tuple τ such that $(C \wedge D)[x/\tau]$ (see footnote[5]) is a logical consequence of the assumed application-specific theory.
- Let C_1 and C_2 be CSPs with the same tuples of free variables. We say that C_1 *refines* C_2 if any solution of C_1 is a solution of C_2.

[5] The notation $(C \wedge D)[x/\tau]$ suggests that a domain element, that is, a nonsyntactic object, be substituted for a variable, which is a syntactic object. Taken literally, this is nonsense. However, there is sound intuition behind this nonsense, as proved by the fact that, if one handles this with care (see e.g. [11,13]), the desired result is achieved anyway.

Curiously, Shoenfield [25] tries to avoid the difficulty by assuming that there is a constant in the language for every domain element. As any reasonable language would have a countable set of names, axiomatization of a domain such as the reals seems to be ruled out.

1.5.2 CSPRs

So far CSPs in general. We continue by considering CSPRs: CSPs over the reals. Here the variables range over the reals. Hence the domains[6] are sets of reals. However, in the interest of practical computer implementation, not all sets of reals can occur as domain: only intervals of reals with floating-point numbers as bounds. These are intervals that are either unbounded (on one or on both sides) or bounded. Any bound that may occur in an interval is one of a finite set of real numbers that can be represented in the floating-point number format of a computer. Such reals are called "floating-point numbers". As the empty set is also counted as an interval, we have that the set of floating-point intervals is closed under intersection. Clearly, for any set S of reals there is a unique least floating-point interval (denoted $bx(S)$) containing S.

In CLP($\mathbf{R}$) we have assumed a conventional theory of the reals. It has equality and inequality as only predicate symbols. It has function symbols for addition, subtraction, multiplication, and division. In a CSP however, we assume that the constraint conjunction is based on a different vocabulary: no function symbols and two additional predicate symbols *sum* and *prod*. Their intended interpretation is such that $prod(x, y, z)$ iff z is the product of x and y, and similarly for $sum(x, y, z)$. Constraint conjunctions of CLP($\mathbf{R}$) can be translated to those of CSPR (usually introducing auxiliary variables) and vice versa. In practice it is desirable to use a theory of reals with additional symbols for commonly used functions such as exponentiation, logarithms, and trigonometric functions.

It is the task of the inference system for CSPRs to obtain a maximally informative ID expression for a given ND formula. We first discuss inference rules. In the section following the next, we present the inference system in which the rules are used.

Inference rules To obtain a maximally informative description, we need to make the domains for the variables small as possible. This can be done by inference rules that transform a CSP into a refinement of it. With every predicate (including at least equality, inequality, sum and product) there is associated a domain-reduction inference rule. We will only show a single example here.

Domain-reduction rule for the prod predicate Let $C \Diamond D_1$ be a CSP where C contains the atom $prod(x, y, z)$ and where the floating-point intervals X, Y, and Z are the projections of D_1 on x, y, and z. The domain-reduction rule for *prod* infers $C \Diamond D_2$ from $C \Diamond D_1$. D_2 is such that all its projections are the

[6] "Domain" means universe of discourse in logic semantics and in the CLP scheme. In constraint processing it has a different meaning: set of values for a variable that are still possible at a certain stage of computation.

same as the corresponding ones of D_1 except possibly for those on x, y, and z, which are shown in the table in Figure 1.1. The operations $*$ and $/$ on intervals are defined in interval arithmetic[7].

Projection on	D_1	D_2
x	X	$bx(X \cap (Z/Y))$
y	Y	$bx(Y \cap (Z/X))$
z	Z	$bx(Z \cap (X * Y))$

Fig. 1.1. Effect of domain-reduction inference rule for *prod* on intervals of CSPR.

If domains were not restricted to floating-point intervals, then the projections of D_2 would simply be $X \cap (Z/Y)$, $Y \cap (Z/X)$, and $Z \cap (X * Y)$. In the interest of implementability we replace these sets by the smallest floating-point intervals containing them. This containment ensures the soundness of the inference rule.

The set $X * Y$ is typically not a floating-point interval, though it is a real interval. This is a mathematical way of saying that rounding errors are typically made when multiplying floating-point numbers. The set Z/Y may not even be a real interval. As inference rules have to yield intervals, these sets have to be converted to intervals. As inference rules have to be sound, the resulting intervals have to contain these sets. Hence the occurrence of the *bx* function in the table of Figure 1.1.

In the following we will repeatedly refer to Cartesian products of intervals. We will call such a product *box*.

Other domain-reduction rules There is a corresponding domain-reduction rule for the *sum* predicate, again based on interval arithmetic. In CSPR, every predicate comes with a domain-reduction rule. Hence also the equality and inequality predicates. These rules are not based on interval arithmetic.

The splitting rule A domain-reduction rule is a directly productive way of achieving our objective: by making a box smaller, it increases information about the location of solutions. But it may happen that no domain-reduction rule has an effect. In that case we turn to the splitting rule an inference rule that is not productive in that sense. It may be indirectly productive by producing CSPs on which domain-reduction rules do have an effect.

The splitting rule replaces $C\Diamond(x \in b)$ by $C\Diamond(x \in b_1)$ and $C\Diamond(x \in b_2)$ such that $b \subset (b_1 \cup b_2)$. We would like to have $b = (b_1 \cup b_2)$ and $b_1 \cap b_2 = \emptyset$, but the limited supply of floating-point intervals may only allow an approximation to this ideal. Both b_1 and b_2 are strictly smaller than b.

[7] Except that modifications are needed accommodate division by an interval containing zero. See [16].

We assume that any given CSPR comes with a *splitting strategy*, with as parameter a real number $\epsilon > 0$. The splitting strategy is a partial function on boxes that, if defined on a box b, yields two boxes b_1 and b_2 as described above, of at least approximately equal size.

Given a box b and such a splitting strategy as partial function, there is a uniquely determined binary tree with b as root where each non-leaf node x has as children the boxes that result from splitting x. The finiteness of the set of floating-point numbers, hence of intervals with floating-point numbers as bounds, hence of finite Cartesian products of such intervals, helps ensure that any splitting strategy's binary tree is finite.

An inference procedure We describe an inference procedure for CSPRs that consists of constructing a search tree with an initial CSPR of the form $C\Diamond(x \in \mathcal{R}^n)$ as root and having as leaf nodes the CSPRs $C\Diamond(x \in b_1)$, ..., $C\Diamond(x \in b_k)$.

We first describe the auxiliary concepts. Then we present an algorithm to define the search tree of the inference system. This algorithm is only suitable for definition and is not intended for execution. Finally, we state what ND/ID sentence is proved by the inference procedure.

NC test A CSP can be sometimes be shown to be unsolvable by showing that it fails to satisfy a Necessary Condition (hence "NC test") for solvability. This is useful because some necessary conditions can be tested with little computational effort. An example in a CSPR of the form $C\Diamond(x \in b)$ is to substitute the projections of b for the corresponding variables in C and then to evaluate C according to interval arithmetic. A necessary condition for solvability of $C\Diamond(x \in b)$ is that this evaluation comes out *true*. This is essentially the "Box(0) consistency" of Puget and Van Hentenryck [23].

Stabilization Every CSP has a fixed repertoire of domain-reduction rules. When these are applied sufficiently many times, a limit CSP is reached that is invariant under all domain-reduction rules. The limit is independent of the order of applying these rules, provided the order is a *fair* one; for details see [20,26,2]. This process is also called *constraint propagation.*

For any given CSP, call it X, constraint propagation to the limit yields a uniquely determined stable CSP. Let us define the *stabilization operator* applied to X as the one that gives this uniquely determined stable CSP.

Search Tree For given NC tests and splitting strategy, the search tree for a CSP is a binary tree of CSPs defined as follows.

1. Obtain the binary tree corresponding to the given CSP and splitting strategy.
2. In the resulting tree, apply to each node the NC tests. If the result is failure, mark the node F.

```
3. Apply the constraint propagation operator to each node not marked $F$.
   If the operator results in failure, then mark the node $F$.
4. while there exists a node $N$ that is not marked $F$
         and that has two successors marked $F$
   do    apply mark $F$ to node $N$
5. while there exists a node $N$ marked $F$
   do    remove subtree rooted at $N$
6. while there exists a node $N$
         with two successors that are leaf nodes
   do    remove these two leaf nodes
```

This algorithm only serves to facilitate the definition of the search tree. Any algorithm intended for execution will of course be much more efficient. The procedure *solve* of BNR Prolog [5] is a simple depth-first traversal of what our search tree would be if we had omitted the last step. Although *solve* is satisfactory in leaving out all failed nodes, it has the shortcoming of reporting adjacent non-failed nodes of what remains of the search tree according to our definition before the last step. This makes *solve* much less useful: it produces long listings of non-failed nodes that turn out to be replaceable by a single one. This is what the last step in our definitional algorithm improves upon.

What does the inference system prove? The above inference system is intended to have the following

Property 1. Let $b_1, \ldots, b_k$ be the boxes at the leaf nodes of the search tree for a CSP of the form $C \Diamond (x \in \mathcal{R}^n)$. Then the ND/ID sentence $\forall x.(C \Rightarrow (b_1 \vee \cdots \vee b_k))$ is a logical consequence of the theory of reals in conjunction with the theory of floating-point numbers.

In principle the proof should be simple: the property claimed holds for the search tree after the first step in the definition algorithm. All that happens in the first step is splitting up the original search space $\mathcal{R}^n$ into sufficiently small boxes without loss of a single point. Each next step, if correctly implemented, removes parts of the search space that do not contain any solution.

An actual proof may, however, be a complex affair. It will have to depend not only on an axiomatization of the reals, but also on certain aspects of floating-point numbers.

1.6 Related work

BNR Prolog [5] has combined Prolog with interval constraints. From the documentation available to us, it is a conventional Prolog where the unsound conventional floating-point arithmetic has been replaced by interval constraints.

For all we know, we may have been introducing Prolog IV. Whether or not that is the case is not easy to tell from the documentation [3] we have studied. The same holds for the Newton system [14]. For either system, presenting it explicitly as a sound CLP(**R**) may only be a formality.

The inference rules are modelled on those of Apt [1] for arithmetic constraints on integers. If his system is interfaced with the CLP scheme the same way CSPRs are in this paper, one would a get reconstruction of the CHIP system [10].

If Newton-like methods were used to enhance the NC tests and the constraint propagation in our inference system, the same search tree would be obtained as in Puget and Van Hentenryck [23]. In addition, they use Brouwer's fixpoint theorem to positively identify intervals containing at least one solution, or exactly one solution. Without such help from outside of the world of constraints one can only eliminate space as not containing any solution, so that answers are always conditional on there existing a solution.

Finally, there are the Russians. Many of the methods of interval constraints on reals were developed in Russia independently of work in the West under the name "subdefinite computation". Sources of pointers to this tradition are [24]. We do not know whether it has been determined who did what first in interval computation.

1.7 Conclusions

We have proposed to extend CLP(**R**) to a sound version by means interval constraints. As mechanism for interfacing the CLP scheme with interval constraint propagation, we proposed a version of Apt's proof-theoretic model for discrete constraint propagation. To do this, it was necessary to propose a logic specification of the example chosen, namely solving a single equation in a single variable. We conjectured that the ND/ID formulas introduced here will serve the same purpose in other numerical problems.

This work brings up a few related more general topics that we wish to discuss in the remainder of this section.

"Logic and numbers don't mix"? There is a widespread perception that logic, and therefore rigorous result verification, belong in the discrete world, which is disjoint from the world of continuous change where variables are real numbers. This perception has had several consequences. At the side of scientific modelling there has been a reluctance to state formally what it means for computer output to be an acceptable solution. At the side of program verification there has been a tendency to stay with discrete problems such as verification of protocols and hardware.

This work has shown that CLP(**R**) can be extended to a sound version. We hope that numerical analysts will find explicit and formal specification of numerical problems a welcome resolution of existing ambiguities.

Floating-point numbers, the untouchables of computer science From a distance, floating-point numbers look like an acceptable surrogate for the reals. Close up they look horrible. There are only finitely many of them. Addition and multiplication, which should be associative, aren't. Multiplication, which should distribute over addition, doesn't.

Conventional numerical analysis has restricted its considerations to algorithms over the reals which are at best heuristic approximations to what happens during computer execution. Interval analysis has shown that with a modest improvement of floating-point hardware, such as achieved by the IEEE standard, one can extend mathematical reasoning to what is actually printed out on an actual computer.

The reluctance to face up to the discrepancies between reals and floating-point numbers has also infected logic programming: CLP($\mathbf{R}$) has ignored it. We have shown that via interval constraints, CLP($\mathbf{R}$) can be extended to cover the entire span from mathematical model in terms of reals at one end to computer output at the other end.

Numerical analysis as an independent science? Van Wijngaarden [27] was one of the few, before and after this 1966 publication, to deplore the ambiguous logical status of results in the conventional practice of numerical computation. As remedy he proposed to replace real analysis by a version more amenable to the idiosyncrasies of floating-point computation. He sketched an axiomatic re-foundation not only of numbers, but also of concepts of analysis such as limit, derivative and integral.

Even if such an ambitious enterprise turns out to be feasible, we doubt that mathematics will be enriched by it. The recent developments discussed in this paper show that it is perfectly practical to use computer hardware to derive true statements about reals that provide exactly the kind of information that engineers and scientists need.

All that was needed for this breakthrough was the following.

1. Use the familiar observation that every function f from a set S to a set T has a counterpart mapping the powerset of S to the powerset of T (the extension of f to the powerset [6]).
2. That most of what one wants to do with f can be done with its extension.
3. Computers, though hopeless at representing reals, are perfectly adequate for representing sets of reals. The limitation of the computer manifests itself in only being able to conveniently represent intervals bounded (if at all) by a floating-point number. Computations can be arranged so that rounding errors only make the set slightly larger. But the set of possible values still contains the true value of the real-valued unknown concerned.

This has been the approach taken by the Russians with their subdefinite computation [24].

1.8 Acknowledgements

Many thanks to Krzysztof Apt for initiatives, encouragement, and suggestions for improvement. The comments of the referee resulted in considerable further improvement. The Natural Science and Engineering Research Council of Canada provided research facilities.

References

1. K.R. Apt. A proof theoretic view of constraint programming. *Fundamenta Informaticae*, 33(3):263–293, 1998.
2. K.R. Apt. From chaotic iteration to constraint propagation. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming (ICALP '97)*, 1997.
3. Frédéric Benhamou, Pascal Bouvier, Alain Colmerauer, Henri Garetta, Bruno Giletta, Jean-Luc Massat, Guy Alain Narboni, Stéphane N'Dong, Robert Pasero, Jean-François Pique, Touraïvane, Michel Van Caneghem, and Eric Vétillard. Le manuel de Prolog IV. Technical report, PrologIA, Parc Technologique de Luminy, Marseille, France, 1996.
4. Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.
5. BNR. BNR Prolog user guide and reference manual. Version 3.1 for Macintosh, 1988.
6. N. Bourbaki. *Théorie des Ensembles (Fascicule de Résultats)*. Hermann, 1939.
7. J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987.
8. A. Colmerauer, H. Kanoui, R. Paséro, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, 1972.
9. E. Davis. Constraint propagation with labels. *Artificial Intelligence*, 32:281–331, 1987.
10. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint programming language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1988.
11. H.B. Enderton. *A Mathematical Introduction to Logic*. Fletcher and Sons, Ltd, 1972.
12. George E. Forsythe. Pitfalls of computation, or why a math book isn't enough. *Amer. Math. Monthly*, 77:931–956, 1970.
13. Andrzej Grzegorczyk. *An Outline of Mathematical Logic: Fundamental Results and Notions Explained with All Details*. D. Reidel, 1974.
14. P. Van Hentenryck, L. Michel, and F. Benhamou. Newton: Constraint programming over nonlinear constraints. *Science of Computer Programming*, 1996.
15. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.
16. T. Hickey, Q. Ju, and M. van Emden. Using the IEEE floating-point standard for implementing interval arithmetic. In preparation.
17. Patricia Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

18. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–582, 1994.
19. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
20. Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.
21. Ramon E. Moore. *Interval Analysis.* Prentice-Hall, 1966.
22. D. Stott Parker. Monte Carlo arithmetic: an effective way to improve upon floating-point arithmetic. Technical Report CSD-970002, Computer Science Department, University of California at Los Angeles, 1997.
23. Jean-François Puget and Pascal Van Hentenryck. A constraint satisfaction approach to a circuit design problem. *Journal of Global Optimization*, 13(1), 1998.
24. Alexander L. Semenov. Solving optimization problems with help of the Unicalc solver. In R. Baker Kearfott and Vladik Kreinovich, editors, *Application of Interval Computations*, pages 211–224. Kluwer Academic Publishers, 1996.
25. Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.
26. M.H. van Emden. Value constraints in the CLP Scheme. *Constraints*, 2:163–183, 1997.
27. A. van Wijngaarden. Numerical analysis as an independent science. *BIT*, 6:66–81, 1966.