Value constraints in the CLP scheme

M.H. van Emden

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`

`http://www-csc.uvic.ca/home/vanemden/vanemden.html`

Abstract

We define value constraints, a method for incorporating constraint propagation into logic programming. It is a subscheme of the CLP scheme and is applicable wherever one has an efficient method for representing sets of possible values. As examples we present: small finite sets, sets of ground instances of a term, and intervals of reals with floating-point numbers as bounds. Value constraints are defined by distinguishing two storage management strategies in the CLP scheme. In value constraints the *infer* step of the CLP scheme is implemented by Waltz filtering. We give a semantics for value constraints in terms of set algebra that gives algebraic characterizations of local and global consistency. The existing extremal fixpoint characterization of chaotic iteration is shown to be applicable to prove convergence of Waltz filtering.

Keywords constraint logic programming, CLP, Waltz filtering, constraint propagation, arc consistency, interval constraints, chaotic iterations, fixpoints

1 Introduction

Consistency methods were developed in Artificial Intelligence to contain combinatorial explosions encountered in scene interpretation [Wal75]. Subsequently the methods were recognized to be of wide applicability [Mac77]. In logic programming, consistency methods were found useful for two difficulties: inefficient search for the solution of combinatorial problems and the ruining of logic semantics by floating-point implementation of real numbers.

The first problem was addressed by CHIP [DHS+88], which incorporated a consistency algorithm by modifying the language of first-order logic, by modifying unification, and by adding inference rules. The success of CHIP has been the main cause of the current interest in constraint logic programming. To address the second problem, BNR Prolog [BNR88, BO] also combined a consistency algorithm with logic programming. The method of combination with the consistency algorithm is different from the one used in CHIP: unification was not modified and no inference rules were added.

A systematic approach to the incorporation of consistency methods in logic programming is to start with the CLP scheme. CLP($\mathcal{R}$), its best-known instance, incorporates algorithmic opportunities such as the Simplex method and Gaussian elimination.

In this paper we find that there is much merit in the systematic approach. The survey paper on the CLP scheme [JM94] contains an indication of how to go about incorporating consistency methods in logic programming by introducing a predicate symbol for each possible domain. As that paper devotes eight lines to the topic (in Example 2.7, which is restricted to the domain of integers), it cannot be expected to be the last word on it. This example gives a hint for the semantic basis for the incorporation of the consistency method. It gives no hint of the required proof theory, nor how constraint propagation algorithms are to be incorporated. In this paper we show that this deficiency can be remedied by a constraint-store management strategy that is different from the one so far used in instances of the CLP scheme. In this way only unspecified aspects of the CLP scheme are elaborated. Hence the result inherits all logical properties of the scheme. In particular, it is based on the clausal form of first-order predicate logic with unmodified unification and resolution as sole inference rule. The resulting instance of the CLP scheme is general enough to cover both finite domains and intervals of reals.

The CLP scheme distinguishes active from passive constraint and is permissive about how these are manipulated. To incorporate the consistency algorithm in the way indicated, we found it necessary to handle the active and passive constraint in a way that is different from how it is done in CLP($\mathcal{R}$), so far the most widely used instance of the CLP scheme. Because of the wide applicability of the consistency algorithm (to combinatorial as well as numerical analysis problems), we view the method presented here as a subscheme of the CLP scheme and we call it *value constraints*.

Approaching the incorporation of the consistency algorithm by first defining a strategy for handling the active and passive constraints in the CLP scheme has the advantage that one rediscovers the consistency method in a very natural way. I show that this not only holds for the general idea, but also for the consistency algorithm itself and several of the algebraic properties found by Benhamou and Older [BO].

Overview of the paper In section 2 we review the CLP scheme. On the basis of this, the next section introduces the notion of store management strategy. Section 4 defines value constraints. As this is presented as a subscheme of CLP, it is defined in terms of logic. To facilitate more detailed development of value constraints, section 5 gives a redescription in terms of algebra. This leads to an improvement treatment of the consistency operator introduced by Benhamou and Older [BO]. Section 6 reviews a result of Lassez and Maher [LM84] on fixpoints of chaotic iterations. This is used in section 7 to derive in a heuristic fashion a simple constraint-propagation algorithm. Concluding remarks are found in section 9.

2 Review of the CLP scheme

Before embarking on the details of the CLP scheme needed for this paper, we first consider reasons for using the scheme.

Exploiting algorithmic opportunities in logic programming Programming languages differ vastly in the sophistication of algorithms used in their implementation. At one extreme we have a precursor of `C++` of which Stroustrup [Str94], page 36, says

> ...that C with Classes "should be implementable without using an algorithm more complicated that linear search."

This is appropriate for *implementation languages*, in which algorithms are implemented. At the other extreme we have languages such as Prolog IV [Col96] and RISC-CLP(Real) [Hon] that are not intended for the implementation of algorithms and which we can therefore classify as *application languages*. These latter require in their implementation certain algorithms that are considerably more complicated than linear search.

Prolog does not fit in this spectrum: it is not suitable for the efficient implementation of algorithms. Yet it does not incorporate sophisticated algorithms in its implementation, although the Warren Abstract Machine is of course a sophisticated abstract machine architecture. The CLP scheme is a way of continuing the language development started by Prolog by making it easier to *exploit algorithmic opportunities*. Prime candidates for these are the Simplex algorithm and Gaussian elimination. These have been incorporated in Prolog III independently of the CLP scheme and via the CLP scheme in CLP(R).

Another important algorithmic opportunity is the Waltz algorithm (also referred to as "constraint propagation," "arc consistency," "filtering"). In one way or another, this has been incorporated in logic programming languages such as CHIP [DHS+88], BNR Prolog [BNR88, OV90, BO], Interlog [BT, Lho93], Newton [BMH94], ICL [LL94], CIAL [CL94], Prolog IV [Col96]. It is the purpose of this paper to use the CLP scheme to exploit this algorithmic opportunity. We find that it is possible to do this at such a fundamental level that, instead of a single instance of the CLP scheme, we get a subscheme with many possible instances.

Details of 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. Hence the scheme has as parameter a tuple $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$, 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 $\mathcal{P}, \mathcal{T} \models \forall[c \to G]$. Here I am particularly interested in the situation where G is a description in logic of a numerical problem, $\mathcal{P}$ contains definitions of predicates occurring in G, $\mathcal{T}$ expresses all we need to know about the real number field, and c uses intervals to give the value of each problem variable accurate to a dozen decimal places. The $\models$ relation guarantees that all these decimal places are correct.

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

The role of C and S in this formula is to describe the answer to the query Q. A favourable property is that $\models$ guarantees correctness of such a description. A difficulty is that the description may not be helpful: it can take the form of a set of equalities and inequalities between polynomials in any number of variables in any degree: a truly formidable problem. In certain types of application, one would like the constraint store to have the form $\{X_1 = r_1, \ldots, X_n = r_n\}$, where $X_1, \ldots, X_n$ are the variables on C or S and $r_1, \ldots, r_n$ are real numbers. However, this is not practically feasible, as the exact values $r_1, \ldots, r_n$ are usually not finitely representable. An excellent surrogate is a constraint store in the form $\{X_1 \in [l_1, u_1], \ldots, X_n \in [l_n, u_n]\}$ with all variables occurring exactly once on the left-hand side and on the right-hand side narrow intervals of reals bounded by floating-point numbers. To bring about this desirable state of affairs is the task of *constraint store management*.

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

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

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 $a = p(s_1, \ldots, s_n)$.

---

[1] $\mathcal{D}$ is a structure 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.

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

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

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 $\langle C', S' \rangle = \mathit{infer}(C, S)$.

4: Consistency test

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

if $\mathit{consistent}(C)$

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

if $\neg\mathit{consistent}(C)$

3 Constraint store management

To understand the distinctive roles of passive versus active constraints, let us first consider derivations from which the transitions $\rightarrow_i$ and $\rightarrow_s$ are excluded. At the end of a successful derivation of this kind, all goals generated directly or indirectly from the query Q have been transformed to an answer constraint only consisting of passive constraints S.

In spite of this restriction, the correctness property for the answer constraints holds: $P, \mathcal{T} \models \forall[S \rightarrow Q]$. Because only $\rightarrow_r$ and $\rightarrow_c$ transitions were used, C is empty and no consistency was ever checked. There are two reasons why such an answer, $\forall[S \rightarrow Q]$, is usually uninteresting: (a) S is typically inconsistent, and (b) getting useful information out of S, even if consistent, is typically a significant computational problem.

Thus the computational problem in the CLP scheme is twofold:

- to show $\mathcal{D} \models \exists S$, consistency of a candidate solution S to the problem posed in query Q. But even in the presence of consistency, it may still be the case that S tells us little about the solution: it may be a difficult numerical problem. Hence,

- to use S to find a formula that describes the solution in a way that is adequate to our purpose.

The CLP scheme introduced active constraints to solve both problems: to facilitate consistency checking and to help describe solutions. By choosing a type of constraint for the active part of the store that facilitates exploiting an algorithmic opportunity, both purposes can be achieved simultaneously.

The CLP scheme requires that the $\rightarrow_i$ transition defined by $\langle C', S' \rangle = \mathit{infer}(C, S)$ has the property

$$\mathcal{D} \models \forall[(C' \wedge S') \leftrightarrow (C \wedge S)].$$

It follows that if the next transition is $\rightarrow_s$ and results in failure, then $\mathcal{D} \models \neg\exists C'$, hence $\mathcal{D} \models \neg\exists(C \wedge S)$; that is, a recent $\rightarrow_c$ transition has caused loss of consistency and the derivation should be halted.

Constraint store management in CLP($\mathcal{R}$). So far constraint store management in general. Let us illustrate this with CLP($\mathcal{R}$). Here the active and passive constraints are conjunctions of atoms. The predicates in constraint atoms are $=$, $\leq$, or $\geq$. The active constraint atoms are all required to be linear: that is, no variable is multiplied by a variable. With such conjunctions the $\rightarrow_s$ step is efficiently implemented by the techniques of numerical linear algebra.

In CLP($\mathcal{R}$), *infer* consists of checking the passive constraints for linearity and transferring any such to the active part of the store. In this way CLP($\mathcal{R}$) exploits an algorithmic opportunity: that linear equalities and inequalities can be efficiently solved. This is achieved by specializing the CLP scheme through a number of choices: $\mathcal{D}$, sublanguage of $\mathcal{L}$ to be used for S and C, *infer* with the implied criterion for distinguishing active and passive constraints, and the algorithm used to implement *consistent*.

A comparison of two strategies for constraint store management. CLP($\mathcal{R}$) depends on identification of an efficiently consistency-testable subset of the passive constraints, which is then removed and added to the active constraints. Active constraints are not redundant. On the contrary: during a derivation the active constraints *replace* the passive ones. In each $\rightarrow_i$ transition, S is depleted as much as possible. I will call this the *replacement strategy* for constraint store management.

In an alternative that we call the *approximation strategy*, the $\rightarrow_i$ transition *leaves S unchanged*. Suppose that a CLP derivation has $\langle C_0, S_0 \rangle, \ldots, \langle C_n, S_n \rangle$ as sequence of $\langle C, S \rangle$ pairs. C_0 is empty. CLP in general requires that $\mathcal{D} \models \forall[C_j \wedge S_j \leftrightarrow C_{j+1} \wedge S_{j+1}]$ and $\mathcal{D} \models \forall[C_{j+1} \rightarrow C_j]$ for $j = 0, \ldots, n-1$.

The replacement strategy has as aim to make S_n empty. If successful, then we have $\mathcal{D} \models \forall[S_0 \leftrightarrow (S_0 \wedge \mathrm{true}) \leftrightarrow (S_0 \wedge C_0) \leftrightarrow (S_n \wedge C_n) \leftrightarrow C_n]$. That is, the final active constraints are equivalent to the initial passive constraint.

The approximation strategy has as aim to make C_n as strong as possible, while keeping the passive constraint store unchanged. Starting from the tautology $S_n \wedge$

$C_n \to C_n$, we can use the general property of CLP to derive $\mathcal{D} \models \forall[S_0 \wedge C_0 \to C_n]$, hence $\mathcal{D} \models \forall[S_0 \to C_n]$. Thus we have only one half of the final result of a successful derivation according to the replacement strategy. But we will see that C_n, though only an approximation of S_0, can be an extremely close one. We will also see that in numerical computation on actual (as opposed to idealized) computers, $\mathcal{D} \models \forall[S_0 \leftrightarrow C_n]$ is not possible in general, whereas $\mathcal{D} \models \forall[S_0 \to C_n]$ is.

4 Value constraints

The *value constraints* subscheme of the CLP scheme consists of the approximation strategy for constraint store management together with the restriction that the active constraint is a conjunction of atoms of the form $v(X)$. As the unary predicate v denotes a subset of D, the active constraints constrain the values allowed for each variable individually, independently of those for any other variable.

The subsets of D denoted by the predicates v occurring in the active constraint may have to be restricted to be efficiently representable in a computer. Three important examples of representable subsets D come to mind.

- D is small and finite, as often happens in combinatorial problems. In this case there is no problem in designating all subsets of D as representable.
- D is the set of real numbers. Now even a single domain element often requires an infinite amount of memory, as do most subsets. Even if we restrict subsets to be intervals, then we cannot represent most of them. A natural choice for computer use is that of *intervals of reals with floating-point numbers as bounds.* It will turn out to be important to include among these intervals the set of all reals and the empty set of reals.
- D is a Herbrand universe with function symbols of positive arity, hence D is infinite. Certain subsets of D are easy to represent even though infinite: those that consist of all ground instances of a term. This term is then a convenient way to represent the subset.

4.1 Definition of the value-constraint subscheme of CLP.

In the CLP scheme the semantic domain is characterized by $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$, where $\mathcal{D}$ is a structure consisting of a set D, the carrier, with functions and relations over D named by symbols in Σ. In the value constraints subscheme of CLP we assume in addition that

1. The passive constraint S is a conjunction of the atoms $A_1, \ldots, A_m$ each of which typically contains few of the many variables $X_1, \ldots, X_n$ that occur in S.
2. The relations in $\mathcal{D}$ include *unary representability relations,* each corresponding to a representable subset of D. We assume that D_{RS}, the set of representable

subsets of D, is closed under intersection in the sense that for any subset X of D_{RS}, the intersection of the elements of X is also an element of D_{RS}. The representable subsets of D are the "privileged subdomains" of Prolog IV [Col96].

Because Σ has a symbol to name each function or relation of $\mathcal{D}$, Σ contains unary predicates v, *value predicates*, to name the unary representability relations.

3. The active constraint has the form

$$C = \{v_1(X_1), \ldots, v_n(X_n)\},$$

where $X_1, \ldots, X_n$, the variables of S, occur each exactly once and where $v_1, \ldots, v_n$ are value predicates, in some enumeration of Σ that depends on C.

4. The consistency test in the value constraints subscheme becomes especially simple: to detect whether any conjunct in the active constraint has a value predicate denoting $\emptyset$.

5. The constraint store management strategy is the approximation strategy.

6. In the CLP transition $\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$, we have $infer(C, S) = \langle C', S\rangle$. In value constraints C' is the result of applying to C the *consistency operator* associated with one of $A_1, \ldots, A_m$. This operator is motivated, defined, and analysed in the remainder of the paper.

   Every time $C' \neq C$ (as we always have $\mathcal{D} \models \forall[C' \rightarrow C]$), the $\rightarrow_i$ makes the active constraints a better approximation to the passive constraints. Hence, $\rightarrow_i$ is repeated until $C' = C$.

The approximation function. Because D_{RS} is closed under intersection, there exists for every subset X of D a unique least representable subset containing X. This universality and uniqueness imply the existence of the function $ap' : \mathcal{P}(D) \rightarrow D_{RS}$ defined as $ap'(X) = \cap\{Y \mid Y \in D_{RS} \text{ and } Y \supset X\}$, where ap' is short for "approximation," following [BO]. The prime indicates that a more important version called ap will be defined later.

4.2 Examples

Small finite sets. Here "small" means that D can be represented by enumerating its elements. Then this is also the case for any subset of D. Hence $D_{RS} = \mathcal{P}(D)$ and ap' is the identity function on $\mathcal{P}(D)$.

Real numbers. D is the set of real numbers. Let M be a finite subset, typically the floating point numbers of a given computer. Then D_{RS} is the set of bounded and unbounded intervals of reals, where bounds have to be in M. ap' maps any set of reals to the least interval in D_{RS} containing it. As the bounds of the intervals are limited to the finite set M, the existence of the *least* interval is easy to see.

Herbrand universes. D is an Herbrand universe. For every term there is an element of D_{RS}, and this element is the set of ground instances of that term. In addition, D_{RS} contains the empty set. The intersection property of D_{RS} is guaranteed by the unifiability (or otherwise) of the corresponding terms. Let S be a possibly infinite set of ground terms. Then $ap'(S)$ is the set of ground instances of the least common anti-instance of the terms in S. This anti-instance was defined by Lassez, Maher, and Marriot in [LMM88]. In this paper they also proved the existence of the required least common anti-instance.

4.3 Local versus global consistency

Ideally, *infer* effects such a transition from C to C' that a change from $v(X_i)$ to $v'(X_i)$ implies that values for X_i are removed that are inconsistent with S in the following sense. Consider first the usual notion of consistency of the passive constraints: $\mathcal{D} \models \exists S$. In keeping with this, we call d a *globally consistent value* for X_i iff $\mathcal{D} \models \exists [S\{X_i/c\}]$, where the constant c names the domain element d. That is, if we fix X_i at c and any values for other variables can be found that make S true in $\mathcal{D}$, then d is a consistent value for X_i. A tuple of values is globally consistent with S iff each component is.

Under the value constraints subscheme we assume that the passive constraint S is a conjunction of atoms, say $A_1, \ldots, A_m$. The property defined above is called *global* consistency because it includes all the atoms simultaneously. Analogously we define d to be a *locally consistent value* for X_i iff we have for all $j = 1, \ldots, m$ that $\mathcal{D} \models \exists [A_j\{X_i/c\}]$, where the constant c names d.

In general, local consistency is weaker than global consistency. If no variable is shared between $A_1, \ldots, A_m$, then the two types of consistency are equivalent. See [Mac77] for the case of binary atoms.

5 An algebraic treatment of value constraints

To further specify the value constraint subscheme, it remains to develop the algorithm for the function *infer* in $\langle C', S' \rangle = infer(C, S)$, as required in the constraint store management step $\rightarrow_i$. Such an algorithm and its properties are best described by changing from the predicate logic language used so far to an algebraic language of sets and relations.

5.1 Relations

To make this paper reasonably self-contained, we discuss in this section the operations on relations used in the sequel. We also need to connect the logic notation for relations with the algebraic one.

The passive constraints as relation. S and $\mathcal{D}$ define together an n-ary relation σ (where n is the number of variables occurring in S) over D consisting of all n-tuples of values for the variables in S that make S true in $\mathcal{D}$. More precisely,

$$\sigma = \{\langle c'_1, \ldots, c'_n\rangle \mid \mathcal{D} \models S\theta \text{ where } \theta = \{X_1/c_1, \ldots, X_n/c_n\} \text{ and } c'_1, \ldots, c'_n \text{ are the objects denoted by } c_1, \ldots, c_n\},$$

assuming that $X_1, \ldots, X_n$ are the variables in S.

Given that we assumed the simple form for S as the conjunction of the atomic formulas $A_1, \ldots, A_m$, it is natural to define in the same way relations corresponding to each of these conjuncts:

$$\sigma_i = \{\langle c'_1, \ldots, c'_{n_i}\rangle \mid \mathcal{D} \models A_i\theta \text{ where } \theta = \{Y_1/c_1, \ldots, Y_{n_i}/c_{n_i}\} \text{ where } c'_1, \ldots, c'_{n_i} \text{ are the objects denoted by } c_1, \ldots, c_{n_i}\},$$

assuming that $Y_1, \ldots, Y_{n_i}$ are the variables in A_i.

The active constraints as relation. Now the active constraints

$$C = \{v_1(X_1), \ldots, v_n(X_n)\}$$

can be given a relational characterization similar to the σ introduced for the passive constraints. Let γ be the relation

$$\gamma = \{\langle c'_1, \ldots, c'_n\rangle \mid \mathcal{D} \models C\theta \text{ where } \theta = \{X_1/c_1, \ldots, X_n/c_n\} \text{ and } c'_1, \ldots, c'_n \text{ are the objects denoted by } c_1, \ldots, c_n\}.$$

These definitions allow us to state in a succinct way that the constraint store has at least one solution:

Proposition 1

$$\mathcal{D} \models \exists(C \wedge S) \quad \textit{iff} \quad \gamma \cap \sigma \neq \emptyset$$

Relations, arities, projections, and joins. The question now arises how σ depends on the σ_i. Analogy with set algebra might suggest that $\sigma = \sigma_1 \cap \cdots \cap \sigma_m$. However, in this more general context this is only valid when each A_i contains all variables. This special case is of no interest here, as we are concerned with the situation where each A_i contains only a few of the many variables occurring in S.

For example, A_1 might contain only X_{13}, X_{40}, and X_{276}, whereas A_2 might contain only X_{33}, X_{40}, and X_{178}. In this case $\sigma_1 \cap \sigma_2$ is of no interest, although composed of two ternary relations. Here we need to distinguish relations according to the more subtle concept of *arity* of a relation, which is the set indexes of variables in the formula from which the relation is derived.

Let J and K be subsets of $\{1, \ldots, n\}$, the set of indexes of the variables of S. If p is a relation of arity J and q is a relation of arity K, then the *natural join* $p \bowtie q$

of p and q is a relation consisting of tuples with components that have their indexes in $J \cup K$. Whenever a tuple s occurs in p and a tuple t occurs in q such that the components with indexes common to J and K are equal, then $p \bowtie q$ contains the tuple with all components copied from s and t. See [Ull88] for the official definition of natural join.

This operation is clearly monotonic in both arguments. An important consequence is that for any relations r_1 and r_2, of any arity, we have $ap(r_1 \bowtie r_2) \subset ap(r_1) \bowtie ap(r_2)$.

The projection $\pi_i(\rho)$ of a relation ρ consisting of n-tuples is the set of all i-th components that occur in any tuple of ρ. Thus $\pi_1(\rho) \times \cdots \times \pi_n(\rho)$ is the intersection of all Cartesian products containing ρ.

5.2 Local and global consistency

For the active and passive constraints we have two similar characterizations:

Proposition 2 $\sigma = \sigma_1 \bowtie \cdots \bowtie \sigma_m$.

Proposition 3 *If C is $v_1(X_1) \wedge \ldots \wedge v_n(X_n)$, then $\gamma = E_1 \bowtie \cdots \bowtie E_n$, where the E_i are the representable sets*

$$E_i = \{c' \mid \mathcal{D} \models v_i(c) \textit{ where } c' \textit{ is the object denoted by } c\}.$$

Because of C's simple structure, this join is a Cartesian product:

$$\gamma = E_1 \bowtie \cdots \bowtie E_n = E_1 \times \cdots \times E_n.$$

Under the value constraints subscheme, the task of *infer* is to change the active constraint in such a way that values for variables are removed that are inconsistent with S.

Proposition 4 *$\pi_1(\sigma) \times \cdots \times \pi_n(\sigma)$ is the set of tuples whose components are values that are consistent with S.*

The form assumed for the active constraint implies that they can only specify relations that are Cartesian products with n components. Thus the best possible approximation of the passive constraint relation σ by the active constraint relation γ is exactly the set of tuples with components that are consistent with the passive constraints. The best we can hope for, then, is that *infer*, possibly in multiple steps, reduces γ to $\pi_1(\sigma) \times \cdots \times \pi_n(\sigma)$.

This ideal cannot be reached for two reasons:

1. Not all of $\pi_1(\sigma), \ldots, \pi_n(\sigma)$ may be representable sets. In other words, it may be that no v exists such that $\mathcal{D} \models v(c) \leftrightarrow c' \in \pi_i(\sigma)$ for all values c' and constants c naming them. Hence the active constraints cannot approach closer than the smallest Cartesian product of *representable* sets that contains σ.

2. No sufficiently efficient algorithm is known that can compute such a smallest product.

Based on the approximation function ap' for sets of single domain values, we define the approximation function ap for n-ary relations defined as

$$ap(\gamma) = ap'(\pi_1(\gamma)) \times \cdots \times ap'(\pi_n(\gamma))$$

for any n-ary relation $\gamma \subset D^n$.

Benhamou and Older [BO] introduced this function and remark that

$$\forall X \subset D^n \; . \; [ap(ap(X)) = ap(X)]$$

(namely, ap is idempotent) and

$$\forall X, Y \subset D^n \; . \; [X \subset Y \Rightarrow ap(X) \subset ap(Y)]$$

(namely, ap is monotonic).

¿From Proposition 4 it follows that

Proposition 5 *$ap(\sigma)$ is the least representable Cartesian product containing the tuples with values consistent with S.*

Thus the relationship $\mathcal{D} \models \forall[S \rightarrow C]$ that characterizes the value constraint subscheme of the CLP scheme, together with the fact that the active constraints can only specify representable Cartesian products, translates to set-theoretic terms as $\gamma \supset ap(\sigma)$. It is desirable, but not in general possible, to have equality in this relation. Let us then design an algorithm for reducing γ to a subset of it that is as close as possible to $ap(\sigma)$.

To reduce γ to $ap(\sigma)$ is to remove from γ values that are not globally consistent with S. We don't have an efficient algorithm for that. But, because of the assumed primitive nature of the relations in the conjuncts $A_1, \ldots, A_m$ of S, we can efficiently remove the tuples from γ that are inconsistent with any A_i individually. Achieving this for all $A_1, \ldots, A_m$ separately is the *local* consistency defined earlier.

The above considerations suggest the following concise characterization of local versus global consistency:

Proposition 6 *Let r be a Cartesian product of arity $\{1, \ldots n\}$.*

$$r \textit{ is globally consistent iff } r \subset ap(\sigma_1 \bowtie \cdots \bowtie \sigma_m)$$

$$r \textit{ is locally consistent iff } r \subset ap(\sigma_1) \bowtie \cdots \bowtie ap(\sigma_m)$$

5.3 The consistency operator

To remove from γ all tuples inconsistent with S, we replace γ by $ap(\sigma)$. Analogy might suggest that to remove from γ all tuples inconsistent with A_i we replace γ by $ap(\sigma_i)$. The analogy is far from perfect because γ and $ap(\sigma_i)$ are in general not of the same arity. Suppose that $X_{i_1}, \ldots, X_{i_{n_i}}$ are the variables occurring in A_i, $X_1, \ldots, X_n$ being the variables occurring in $S = \{A_1, \ldots, A_m\}$. Then the arity of $ap(\sigma_i)$ is $\{i_1, \ldots, i_{n_i}\}$, whereas the arity of γ is $\{1, \ldots, n\}$. Thus, γ and $ap(\sigma_i)$ do not have in general the same arity, so that $\gamma \cap ap(\sigma_i)$ is not an interesting relation. However, $\gamma \bowtie ap(\sigma_i)$ is.

Proposition 7 *$\gamma \bowtie ap(\sigma_i)$ is the smallest representable Cartesian product containing the result of removing from γ all tuples that are not consistent with the conjunct A_i of the passive constraint S.*

Proof. Both γ and $ap(\sigma_i)$ are Cartesian products, hence completely characterized by their projections. For Cartesian products the join operation is particularly simple: $\pi_j(\gamma \bowtie ap(\sigma_i)) = \pi_j(\gamma) \cap \pi_j(ap(\sigma_i))$ if X_j occurs in A_i and $\pi_j(\gamma \bowtie ap(\sigma_i)) = \pi_j(\gamma)$ otherwise. $\pi_j(ap(\sigma_i))$ is the smallest representable set containing the values for X_j that are consistent with A_i. Hence $\pi_j(\gamma \bowtie ap(\sigma_i))$ is the result of removing from $\pi_j(\gamma)$ all values for X_j that are not consistent with A_i. $\square$

With this in mind it is clear that removing from γ all tuples inconsistent with σ, while still retaining a representable product, results in $\gamma \bowtie ap(\sigma)$. Because of the special case that γ and σ have the same arities, we happen to have $\gamma \bowtie ap(\sigma) = \gamma \cap ap(\sigma)$. Benhamou and Older [BO] only use $\cap$, so that their treatment covers only the case in which every constraint contains all variables so that they cannot distinguish between local and global consistency.

The expression $\gamma \bowtie ap(\sigma_i)$ has some special features: (a) the arity of $ap(\sigma_i)$ is contained in the one for γ, and (b) both arguments of the join are Cartesian products. The role of the expression together with its special features suggest that one introduces the *consistency operator* K indexed by an arbitrary relation p and acting on a Cartesian product q defined by $K_p(q) = ap(p) \bowtie q$, where $\text{arity}(p) \subset \text{arity}(q)$.

Theorem 1 *For all Cartesian products u and v of equal arity including the arity of relation p we have:*

1. $K_p(u) \subset u$ *(Contractance)*
2. $u \bowtie p = K_p(u) \bowtie p$ *(Correctness)*
3. $u \subset v \Rightarrow K_p(u) \subset K_p(v)$ *(Monotonicity)*
4. $K_p(K_p(u)) = K_p(u)$ *(Idempotence)*

Proof. We apply the definition of K_p and use properties of the natural join.

1. $K_p(u) = u \bowtie ap(p) \subset u$
2. $K_p(u) \bowtie p = u \bowtie ap(p) \bowtie p = u \bowtie p$ because $p \subset ap(p)$ hence $ap(p) \bowtie p = p$.
3. By monotonicity of natural join.
4. As both sides are Cartesian products, we only need to consider the projections: $\pi_j(K_p(K_p(u))) = \pi_j(ap(p) \bowtie (ap(p) \bowtie u))$ (definition K_p) $= \pi_j(ap(p)) \cap \pi_j(ap(p) \bowtie u)$ (both sides Cartesian products) $= ap'(\pi_j(p)) \cap ap'(\pi_j(p)) \cap \pi_j(u))$ (applying the definition of ap)$= ap'(\pi_j(p)) \cap \pi_j(u)) = \pi_j(ap(p) \bowtie u) = \pi_j(K_p(u))$

□

In case p and q have the same arity, $K_p(q) = q \bowtie ap(p)$ specializes to $K_p(q) = q \cap ap(p)$. Benhamou and Older [BO] invented the operator for this case and called it the "narrowing operator" — it contracts q to $K_p(q)$. For the case they considered, they found the properties stated in the theorem. We adopt their names for the properties. Because of their restriction to equal arities, the narrowing operator does not distinguish local from global consistency. In fact, their treatment only applies to the untypical case where all of $A_1, \ldots, A_m$ contain all the variables $X_1, \ldots, X_n$.

Examples. If the constraint atom is $X_1 = X_2$, then the consistency operator reduces $v_1(X_1)$ and $v_2(X_2)$ in the active constraint to $v(X_1)$ and $v(X_2)$ where v denotes the intersection of the denotations of v_1 and v_2. This is true whatever D is.

Suppose that $\mathcal{D}$ is $\mathcal{R}$ and that D_{RS} is the set of bounded and unbounded intervals with machine numbers as bounds. If the constraint atom is $X_1 + X_2 = X_3$, then its consistency operator reduces $v_1(X_1), v_2(X_2)$ and $v_3(X_3)$ in the active constraint to $v_1'(X_1), v_2'(X_2)$ and $v_3'(X_3)$. Let v_1, v_2 and v_3 denote the intervals $[x_1, x_2]$, $[y_1, y_2]$, and $[z_1, z_2]$ respectively. Then it may be shown that the denotation of v_1', v_2' and v_3' is, respectively,

$[x_1, x_2] \cap ([z_1, z_2] - [y_1, y_2])$,
$[y_1, y_2] \cap ([z_1, z_2] - [x_1, x_2])$, and
$[z_1, z_2] \cap ([x_1, x_2] + [y_1, y_2])$.

The operation "$-$" in the above formula is extended from the reals to intervals as follows:

$$[a, b] - [c, d] = [(a - d)^-, (b - c)^+].$$

The a, b, c, and d are machine numbers, so that $[a, b]$ and $[c, d]$ are representable sets (in D_{RS}). In the right-hand side, the superscripts "$^-$" and "$^+$" indicate that the usually inevitable rounding in computing $a - d$ and $b - c$ is done downward and upward, respectively. Thus $[(a - d)^-, (b - c)^+]$ is also a representable set and it is the smallest such that contains the usual canonical extension of $[a, b] - [c, d]$, which is $\{x - y \mid x \in R \wedge x \in [a, b] \wedge y \in R \wedge y \in [c, d]\}$.

A similar discussion goes with $[a, b] + [c, d] = [(a + c)^-, (b + d)^+]$. These are the usual formulas for interval arithmetic [Moo66]. The remarks about directed rounding are from Hansen [Han92], though there are probably earlier treatments of this topic in the interval arithmetic literature.

6 Efficiently computable approximations to a consistent state

The relations defined by the active and the passive constraint are γ and $\sigma = \sigma_1 \bowtie \cdots \bowtie \sigma_m$ respectively. We would like to remove all inconsistent values from γ and this would result in γ becoming equal to $\gamma \bowtie ap(\sigma)$, for which we lack an algorithm. But the primitive nature of the passive constraints $A_1, \ldots, A_m$ allows us to efficiently compute $K_{\sigma_i}(\gamma)$, which was defined as $\gamma \bowtie ap(\sigma_i)$.

Thus we repeatedly compute $\gamma \leftarrow K_{\sigma_i}(\gamma)$ for various values of i. It will turn out to be important that we are as free as possible in the choice of i at each stage. Lassez and Maher [LM84] studied such, what they called, *chaotic iterations*. We are interested in conditions under which chaotic iterations converge to $K_\sigma(\gamma)$. Such an iteration may be finite or not.

The goal of the iteration is the same as that of what Davis [Dav87] calls the "Waltz Algorithm." The contribution of Waltz is a technique for exploiting the freedom in selecting the i so as to expedite convergence. In this paper we will derive this technique for the more general situation considered here.

6.1 Fixpoint theory of chaotic iterations

In the iteration of repeatedly replacing γ by $K_{\sigma_i}(\gamma)$ we think of γ as a *state*[3] (of approximation to $K_\sigma(\gamma)$). We think of i as selecting one of m available operators. As the states are relations, they form a lattice with set inclusion as partial order.

Lassez and Maher [LM84] have given a fixpoint theory of chaotic iterations. Their purpose was to model SLD resolution as a chaotic iteration. We review these results here.

Lassez and Maher assume a complete lattice L (with partial order $\preceq$) and monotonic closure operators $p_i : L \rightarrow L$, for $i = 1, \ldots, m$. A closure p satisfies $p(x) \succeq x$ and $p(p(x)) = p(x)$ for all $x \in L$. Monotonicity of p is defined as $x \preceq y$ implies $p(x) \preceq p(y)$ for all x and y in L. For such a set of closures they obtain the proposition that for any lattice element x there exists a unique least one among the fixpoints common to all of $p_1, \ldots, p_m$ that are above x in the lattice ordering.

Lassez and Maher also consider a method for computing this fixpoint. Let $f : \omega \rightarrow \{p_1, \ldots, p_m\}$, where ω is the set of natural numbers. Hence f is an infinite sequence of operators. As we regard the function f as a sequence, we write f_n instead of $f(n)$. Let f be a *fair* sequence in the sense that for any natural number N and for any $i = 1, \ldots, m$ there exist infinitely many $n \geq N$ such that $f_n = p_i$.

A lattice element s_0 and a sequence f together define $s : \omega \rightarrow L$ to be a sequence of lattice elements where $s_i = f_{i-1}(s_{i-1})$ for $i = 1, 2, \ldots$ Thus s is the result of applying the operators $p_1, \ldots, p_m$ to an initial lattice element s_0 in an unspecified order except for the condition that each operator keeps recurring, as made precise by the definition

[3] Not to be confused with the states $\langle A, C, S \rangle$ that occur in CLP derivations.

of fairness. Because $p_1, \ldots, p_m$ are closures, s is a nondecreasing sequence. By lattice completeness, $\bigvee s \in L$.

Let z be a fixpoint common to all of $p_1, \ldots, p_m$ such that $s_0 \preceq z$. The question arises naturally: What is the relation between $\bigvee s$ and z?

By monotonicity, $s_n \preceq z$ for all $n \in \omega$. By the definition of $\bigvee$, $\bigvee s \preceq z$. If $\bigvee s$ is a itself a common fixpoint of $p_1, \ldots, p_m$, then it is the unique least fixpoint above s_0 that exists by the proposition. Whether this is the case, depends on the application.

This summarizes a few of the many results in Lassez and Maher [LM84]. In this paper, we have a situation dual to the one considered in [LM84]: the states of the active constraint store form a lattice and $K_{\sigma_1}, \ldots, K_{\sigma_m}$ are the *duals* of closures: besides being idempotent, they are *contractant*: $K_{\sigma_i}(x) \preceq x$ for all lattice elements x. Translating the above proposition to its dual we conclude that $K_{\sigma_1}, \ldots, K_{\sigma_m}$ have a unique *greatest* common fixpoint that is *under* s_0 in the lattice ordering.

Let z be a fixpoint common to all of $K_{\sigma_1}, \ldots, K_{\sigma_m}$ such that $s_0 \succeq z$. By monotonicity, $s_n \succeq z$ for all $n \in \omega$. By the definition of $\bigwedge$, $\bigwedge s \succeq z$. If $\bigwedge s$ is a common fixpoint of $K_{\sigma_1}, \ldots, K_{\sigma_m}$, then it is the unique greatest fixpoint below s_0 that exists by the proposition.

Theorem 2 *If D_{RS} is finite, then $\exists N \in \omega$ such that s_N is the greatest fixpoint common to all consistency operators that is less than s_0.*

Proof. We assumed that D_{RS} is finite, so that the lattice L is finite. As a result, s, which is now a nonincreasing sequence, is constant from a certain index onward: $\exists N \in \omega$ such that $s_n = s_N = \bigwedge s$ for all $n \geq N$.

By the fairness of f, $\forall i$ such that $i = 1, \ldots, m$, we can select an n greater than N such that $s_n = p_i(s_{n-1}) = p_i(\bigwedge s)$. Hence $\bigwedge s = p_i(s_n) = p_i(p_i(\bigwedge s)) = p_i(\bigwedge s)$, by the idempotence of p_i and $\bigwedge s$ being the limit. Therefore $\bigwedge s$ is a common fixpoint, it is the unique greatest common fixpoint that is under s_0, and it is reached after N steps. □

Any fair sequence of operators causes convergence after a finite number of steps. Clearly, fairness is the weakest possible condition on the sequence f. If for any $i = 1, \ldots, m$, p_i would have a last occurrence in f, then examples can be found where the sequence converges to a fixpoint common to the other operators that is not a fixpoint of p_i.

Older and Vellino [OV93] derive the result independently of [LM84] for a more restricted class of fair sequences f, which consists of repetitions of a single permutation of $\{1, \ldots, m\}$.

7 The Waltz algorithm as chaotic iteration of consistency operators

To summarize section 6, if the number of representable sets is finite, then for *any* fair sequence f of operators selected from $K_{\sigma_1}, \ldots, K_{\sigma_m}$, the corresponding sequence s of states converges to the same limit that depends only on s_0 and $K_{\sigma_1}, \ldots, K_{\sigma_m}$. Given this characterization, let us consider an algorithm to compute the greatest common fixpoint.

As this fixpoint is reached in a finite number of steps, and the algorithm should terminate as soon as possible after, we are only concerned with a finite prefix of the fair sequence f. Although the concept of fairness does not apply to this prefix because of its finiteness, it is still useful to think of the algorithm as constructing a prefix to an infinite sequence s of states defined by a fair sequence f of operators.

We classify occurrences of operators in f according to whether they are *live*, *dormant*, or *extinct*.

An operator is

- *live* if it changes the state,
- *dormant* if it does not change the state, and
- *extinct* if it does not change the state and if the same holds for all later occurrences of the operator,

In designing an algorithm, we aim for sequences where the limit occurs as early as possible. It helps, for example, to avoid the application of dormant operators. In the typical situation, where each constraint involves only a small subset of all variables, the fair sequence obtained from ordering the constraints according to a permutation of their indexes, and then repeating this same permutation forever, as considered in [OV93], has the disadvantage that unnecessarily many dormant operators are applied.

For, whenever a live operator is applied, it always becomes dormant, by the idempotency of the consistency operator. It will surely remain dormant as long as no other operator has contracted the set associated with at least one of its variables. An algorithm should therefore postpone application until such contraction has occurred. That will not, of course, guarantee liveness. The only way to determine liveness with certainty is to actually apply the operator. On the other hand, at each index we can generally tell for sure that certain constraints are dormant.

The above considerations suggest that an algorithm maintain a candidate set that includes all live operators and excludes the operators of which one can be sure that they are not. To ensure that the candidate set include all live operators, it seems inevitable to include some that are not.

The algorithm consists of a loop of which the invariant is the property of the candidate set just mentioned. It is initialized by containing all constraints. Each time around the loop, while the candidate set is nonempty, an operator K_{σ_j} is applied and removed from the candidate set because it is now dormant. Any dormant K_{σ_i} such that A_i contains a variable whose domain was changed by applying K_{σ_j}, is added to the candidate set, as this constraint may have turned live.

This suggests the program in Figure 1.

$C \leftarrow$ the set of all operators;
while C not empty
    do remove operator K from C;
        $\gamma' \leftarrow K(\gamma)$;
        for $i \leftarrow 1$ **to** m **do**
            if $\pi_i(\gamma') \neq \pi_i(\gamma)$
            then for each operator K' of a conjunct containing variable X_i
                do add K' to C unless $K' \in C$
        $\gamma \leftarrow \gamma'$
    od

Figure 1: The Waltz algorithm.

Partial correctness is guaranteed by the invariant. Let us now consider termination. Under our assumption of a finite set of representable sets, any infinite sequence of states has to remain unchanged from a certain index onwards. After this index, all operators are extinct. Therefore, whenever an operator is applied, no operator is returned to the candidate set. After at most m times around the loop, the candidate set is empty so that the algorithm terminates at the latest at m steps after the limit has been reached.

8 Related work

This paper can be decomposed into a syntactic part, introducing the notion of value constraints, and a part that presents constraint propagation in such a way that it can serve as semantics for value constraints. We will discuss related work according to this dichotomy.

Related work on logic-based systems The incorporation of constraint propagation in logic programming languages has, so far, not been done via the CLP scheme.

CHIP [DHS+88] has achieved the desired goal by a number of modifications. The language of first-order logic was modified to include a variable for each representable subset. Unification was modified as well. Several inference rules were added.

BNR Prolog [BNR88, BO] incorporated constraint propagation without modifying the language and using the same unification and resolution as Prolog or the CLP scheme. In fact, value constraints can be viewed as an attempt to define an instance of the CLP scheme as close as possible to BNR Prolog.

Prolog IV [Col96] presents the user with a model that is quite different from the CLP scheme. It has in common with instances of the CLP scheme that its domain is an explicitly defined structure in the sense of logic that is not a Herbrand universe and that its language is a first-order language of predicate logic. Whereas CLP scheme instances have been built on conventionally accepted structures, Prolog IV is based on a novel, custom-built structure named π_4. In Prolog IV the structure is partially characterized by a theory $\mathcal{T}(\pi_4)$, consisting of axioms. Given a user's constraint p, computation consists in finding a constraint q that makes the solutions of p, if any, more "visible" and that satisfies

$$\mathcal{T}(\pi_4) \models p \equiv q$$

.

The axioms in $\mathcal{T}(\pi_4)$ are organized as 25 axiom schemas. They take the form of equivalences, but are used as ordered rewrite rules that replace instances of left-hand sides by instances of right-hand sides. This ordering is designed to help in making solutions, or their absence, more visible. These axioms cover every computation step of a logic programming language with constraints and with constraint propagation. Thus, for example, each individual unification step is effected by an axiom specialized for this purpose, so is goal reduction, and so on. As another example, the effect of what we call the consistency operator K_p associated with, say, the ternary predicate p is achieved by applying an axiom of the form

$$\forall x, y, z.p(x, y, z) \wedge A(x) \wedge B(y) \wedge C(z) \equiv p(x, y, z) \wedge A'(x) \wedge B'(y) \wedge C'(z)$$

where the box determined by A', B', and C' is the one resulting from applying K_p, the consistency operator associated with p, to the box determined by A, B, and C.

Thus all the transitions of the CLP scheme have equivalents among the axioms of Prolog IV. In view of such an equivalence, one may wonder what use the CLP scheme might have. A way to do so is to imagine that only Prolog IV had been invented, but not CLP.

The user of Prolog IV is presented with 12 rewrite rules, 25 axiom schemas, and 105 built-in relations (not counting the special functions). Although understanding the axiom schemas is made easier by a classification into eight categories, it is still a complex model for the user. The complexity is unavoidable, given the unprecedented range of facilities offered by this language. But one may be able to help the user by identifying simpler sublanguages.

One such sublanguage is Prolog[4]. Prolog is clearly identifiable in every instance of the CLP scheme. One may even define an instance that has nothing else. However, Prolog is not clearly identifiable in Prolog IV. It seems that a rough approximation can be obtained by only considering the rewrite rules and axiom schemas 4, 5, and 6, but to do so one would have to sweep many details under the rug.

Another candidate for adding structure to Prolog IV is the general idea of constraint propagation. In Prolog IV the general idea is only present in the form of its detailed implementation by means of the individual axiom schemas. The value

[4]In this rough thought experiment we neglect differences such as the one between (1) Prolog with finite trees and with the occurs check and (2) Prolog with rational trees and with Prolog-II type unification.

constraints subscheme of CLP presented here uses the *infer* step of the CLP scheme to encapsulate everything to do with constraint propagation.

All that a system such as Prolog IV can do for the user is to rewrite the initial constraints to a form that is "more visible", in the words of [Col96]. A more detailed description of this concept is helpful to user and implementer alike. The active constraints in CLP, and especially in the value-constraints subscheme, are a useful way to make the notion of visibility more concrete for the class of applications that use constraint propagation.

The above remarks about the complexity of Prolog IV should not be taken as criticism. That would only be appropriate if there existed an equally powerful language that has been presented in a simpler way. As far as we know, this is not the case.

What the world also needs is a framework that does two things: to make it easier to understand a language such as Prolog IV and to make the incorporation of new domains and new algorithms a routine matter. The CLP scheme has this potential, with help of some minor additions such as the value constraints subscheme. The past has presented us with two disparate developments: the CLP scheme, and, independently of that, a sequence of superior implementations, Prolog II, Prolog III, and Prolog IV that could have been in the CLP framework, but weren't. It is to be expected that in the future the framework and the superior implementation are no longer disjoint.

Related work on constraint propagation algorithms The purpose of this paper is to extend the algebraic characterization of the consistency method given by Benhamou and Older [BO]. Inadvertently perhaps, they have only characterized constraint systems where each constraint involves all variables. In this way the distinction between local and global consistency vanishes. Here we show that a minor modification, using a join instead of intersection, can bring their technique to bear on constraint propagation in its full generality.

The version of the Waltz algorithm given here is only interesting as the end point of a heuristic derivation starting all the way back in value constraints. As such, one cannot expect it to be a state-of-the-art constraint satisfaction algorithm. For a 1991 state of the art, see Montanari and Rossi [MR91]. This paper contains also several valuable references. In common with most literature in the West, however, it omits references to the independently developed Russian school of constraint propagation techniques. Recent publications in this tradition are [TU96, Sem96], where references going back to 1980 can be found.

9 Conclusions

In this paper we have taken the logic programming-centered view: the starting point was to see how consistency methods could solve problems in logic programming. This point of view may be too parochial, as consistency methods are extremely powerful in their own right. There are indications that interval constraints, for example, can solve certain numerical problems as well or better than the approach via numerical analysis [BMH94, Old94, Hic94, CvE]. This gives constraint solving a much greater significance than the role they play in the CLP scheme, which is to generalize unification and to describe answers to queries.

Given that constraint systems are as important in their own right as, say, all of logic programming, one may well wonder whether the two are necessarily related.

When considering how to use a constraint system by itself, one must realize that the constraint formula must have the simple form of a conjunction of atomic formulas where the predicate symbols name relations sufficiently simple for the consistency operator to be efficiently computed. In a numerical setting, for example, these relations are typically ternary sum, ternary product, $=$, $<$, $\leq$, power with integer exponent, and ternary min and max. The user may input equalities and inequalities between polynomials in many variables occupying a few lines, yet representing a difficult problem. To obtain the help of a constraint system, these equalities and inequalities need to be translated to the primitive constraints mentioned. The result can easily be hundreds of atomic formulas, introducing hundreds of auxiliary variables. Moreover, the consistency method typically leaves intervals that are too large, so that alternative constraint systems have to be efficiently generated and subjected to the Waltz algorithm.

This indicates that a constraint system needs a programming language as front end. For some constraint systems, this is `C++`. For BNR Prolog, it is Prolog. Does logic programming only connect to constraint systems by Prolog being a front end programming language preferred by certain users? We believe that there is a stronger connection. Here we only attempt to make the case for it in the numerical domain.

Starting with Moore [Moo66], the interval community within numerical analysis developed a method for guaranteed bounds for numerical results. These bounds have the practical virtue of being valid on actual rather than idealized computers. This great achievement was stymied by two problems.

The first is that interval arithmetic often gives grossly pessimistic bounds. Thus a necessary complement are methods to reduce interval width. Such methods have been developed in interval arithmetic. An advantage of bringing interval arithmetic under consistency methods as a special case is that additional methods for reducing interval width become available [CvE].

Another problem of interval arithmetic is the use of programs written in conventional languages, where program verification is not practicable. The guarantees about numerical results that are in principle possible by means of interval methods only carry full force if one can verify the correctness of the program that executes the interval operations. For this reason it is not only of great advantage to incorporate interval methods in a logic programming language, but it is also important to investigate in full detail the connection between the logic and the interval consistency algorithm. This paper shows that such a connection can be made via the CLP scheme.

A radical approach toward correctness in computation is to ensure that computation is a logical deduction. This idea was introduced by Cordell Green [Gre69] and

Patrick Hayes [Hay73]. The latter coined the motto:

Computation is controlled deduction.

The programming language Prolog comes close to supporting this paradigm. But it only comes close in symbolic computation. Floating-point arithmetic is handled just as poorly as in any conventional language. Prolog IV shows that even numerical computation can be deduction, even on actual, rather than idealized, computers. In this paper we have presented a framework that can help the further development of languages like Prolog IV.

10 Acknowledgements

I gratefully acknowledge my indebtedness to Krzysztof Apt for suggesting the topic of this paper and for arranging a visitorship at CWI, the Centre for Mathematics and Informatics in Amsterdam. To Bill Older I am indebted more than the references to his work can suggest. He supported me with many discussions and explanations and provided me with numerous unpublished notes. The anonymous referees have made many valuable suggestions for improvement. Encouragement from Frédéric Benhamou has helped the paper through certain crucial moments. Finally, the Natural Science and Engineering Research Council of Canada generously provided research facilities.

References

[BMH94] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proc. 1994 International Symposium*, pages 124–138, 1994.

[BNR88] BNR. BNR Prolog user guide and reference manual. 1988.

[BO] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming.* Technical report 1994; to appear 1997.

[BT] B. Botella and P. Taillibert. INTERLOG: constraint logic programming on numeric intervals. In *Third International Workshop on Software Engineering, Artificial Intelligence, and Expert Systems, Oberammergau, 1993.*

[CL94] C.K. Chiu and J.H.M. Lee. Towards practical interval constraint solving in logic programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 109–123, Ithaca, USA, 1994.

[Col96] Alain Colmerauer. Les bases de Prolog IV. Technical report, PrologIA, Marseilles, France, 1996. Chapter 2 of "Le Manuel de Prolog IV".

[CvE] H.M. Chen and M.H. van Emden. Interval constraints for unconstrained global optimization. In preparation.

[Dav87] E. Davis. Constraint propagation with labels. *Artificial Intelligence*, 32:281–331, 1987.

[DHS+88] 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.

[Gre69] C.C. Green. The application of theorem-proving to question-answering systems. Technical Report 8, Artificial Intelligence Group, Stanford Research Institute, 1969.

[Han92] Eldon Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, 1992.

[Hay73] P.J. Hayes. Computation as deduction. In *Proc. Second MFCS Symposium*, pages 105–118. Czechoslovak Academy of Sciences, 1973.

[Hic94] Timothy J. Hickey. CLP(F) and constrained ODE's. In Jean Jourdain, Pierre Lim, and Roland H.C. Yap, editors, *Workshop on Constraint Languages and their Use in Problem Modelling*, pages 69–79, Ithaca, New York, November 1994.

[Hon] Hoon Hong. RISC-CLP(Real): Logic programming with non-linear constraints over the reals.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–582, 1994.

[Lho93] O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.

[LL94] J.H.M. Lee and T.W. Lee. A WAM-based abstract machine for interval constraint logic programming. In *Proceedings of the Sixth IEEE International Conference on Tools with Artificial Intelligence*, pages 122–128, New Orleans, USA, 1994.

[LM84] J.-L. Lassez and M.J. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, 29:167–184, 1984.

[LMM88] J-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Moo66] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[MR91] Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[Old94] William J. Older. Application of relational interval arithmetic to ordinary differential equations. In Jean Jourdain, Pierre Lim, and Roland H.C. Yap, editors, *Workshop on Constraint Languages and their Use in Problem Modelling*, pages 60–69, Ithaca, New York, November 1994.

[OV90] W. Older and A. Vellino. Extending Prolog with constraint arithmetics on real intervals. In *Proceedings of the Canadian Conference on Computer & Electrical Engineering*, Ottawa, Canada, 1990.

[OV93] William Older and André Vellino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming*, pages 175–195. MIT Press, 1993.

[Sem96] 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.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

[TU96] Vitaly Telerman and Dmitry Ushakov. Subdefinite models as a variety of constraint programming. In *Proc. 8th Int. Conf. on Tools with Artificial Intelligence*, pages 157–163. IEEE Computer Society, 1996.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

[Wal75] D. Waltz. Understanding line drawings in scenes with shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.