Constraint Logic Programming - An Overview

Fruehwirth Thomas *
Christian Doppler Laboratory for Expert Systems
Paniglgasse 16/E181-2
A-1040 Vienna/Austria
thomfr@vexpert.at

8th August 1990

Abstract

In this report we give an overview of Constraint Logic Programming based on the available work as mentioned in the bibliography. Constraint logic programming languages are a new, powerful class of programming languages based on mathematical logic which are extended in a logically sound way by constraint solving techniques. The result are highly declarative and flexible languages, which are well suited for combinatorical search problems and linear arithmetic equation solving, features useful in application areas like planning and scheduling, circuit design and operations research. After introducing the basic computation domains, namely numbers and boolean values, we discuss current constraint logic programming languages in detail. In this way an overview of the state-of-art in constraint logic programming is given and its potential applications are outlined through examples.

1 Introduction

During the 1980s we have seen the rise of a new programming paradigm called *logic programming*. The most prominent representative of this new programming paradigm is the language *Prolog*, developed in the early 1970s by Colmerauer in Marseille and Kowalski in Edinburgh. Programming in Prolog differs from conventional programming both stylistically as well as computationally, as it uses logic to represent knowledge and deduction to solve problems. Due to the success of Prolog in the academic world, logic programming today slowly begins to find its way out of the research labs into advanced products like expert systems or knowledge-based systems.

*This work has been supported by the Austrian Industries Holding

The recent profileration of *extensions* to logic programming reflects, on one hand, the popularity of these languages, and, on the other hand, their limitations. Among the proposed extensions to Prolog, several versions incorporate other paradigms and/or languages. These include Loglisp (Lisp and Prolog), Funlog (Functional Programming and Prolog), and Eqlog (Term Rewriting and Prolog). The question arose, however, whether these extensions preserved the logical basis of the language. In many cases, the answer was negative.

It has been argued in the literature that a program is best divided into two components called *competence* and *performance*. The competence component contains factual information - statements of relationships - which must be manipulated and combined to compute the desired result. The performance component then deals with the strategy and tactics of the manipulations and combinations. The competence part is responsible for the correctness of the program; the performance part is responsible for the efficiency and termination. We would like, in our programming, to concern ourselves first with competence (*"what"*), and only then, if at all, worry about performance (*"how"*). Logic programming provides a means for separation of these concerns. It is based on *first order predicate logic*, and the performance component is mostly automatic by relying on a built-in computation mechanism called *SLD-resolution*.

In this way, logic programming has the unique property that its *semantics*, operational and declarative, are both simple and elegant and coincide in a natural way. This property, however, comes at a price. The semantics of a logic program are defined within the context of the *Herbrand Universe* - the set of all possible terms that can be formed from the functions and constants in a given program. In this universe, only those terms which are syntactically equivalent can be unified together. Every semantic object has to be explicitely coded into a Herbrand term; this enforces reasoning at a primitive level.

On the other hand to *implicitely* describe the objects of discourse, *constraints* are widely used in applications such as engineering, knowledge representation, and graphics. Additionally, many real life problems like scheduling, allocation, layout, fault diagnosis and hardware design can be seen as constrained search problems. Constraint manipulation and propagation have been studied in the Artificial Intelligence community in the late 1970s and early 1980s especially in the USA [L80, Wm88, FBBNA90, ea88a]. They provide problem solving techniques like local value propagation, data driven computation, forward checking (to prune the search space) and consistency checking. The most common approach for solving a given constraint problem consists in writing a specialized program in procedural languages. This approach requires substantial effort for program development, and the resulting programs are hard to maintain, modify and extend.

Constraint Logic Programming is an attempt to overcome this difficulties by providing declarativeness and flexibility by enhancing a Prolog-like language with a constraint solving mechanism. Not only does this free the logic programmer from the restrictions of the Herbrand Universe, it also enables to increase

efficiency and expressability by using special purpose *constraint solvers* over specific domains. A constraint solver is an algorithm deciding the satisfiability of constraint systems.

A *constraint* in logic programming is viewed as a special predicate, i.e. a relation that should be satisfied. Placing a constraint that the quantity named a is less than the quantity named b means that there is a known relationship between the two. Similarily, if the sum of three values x, y and z is constrained to be zero, then this relationship can be viewed in more than one way: For example, one might find convienient for some purposes the view that x is minus the sum of the other two. In other words, constraints are *multi-directional*. Constraint-based languages allow the user to state declaratively a relation that is to be maintained, rather than requiring them to write procedures to maintain the relations themselves.

The rest of the paper is organized as follows: In the next section we will introduce two important *computation domains*, namely numbers and boolean values, including some classical examples. Then we give an overview over current constraint programming languages. Last but not least the appendix presents the overheads of a talk on constraint logic programming given for the CD-Lab in May 1990. In particular, it offers a different introductory view on constraint logic programming and introduces additional examples.

2 Computation Domains

In this section, we introduce two basic computation domains for constraints, namely numbers for *linear arithmetic* and truth values for *boolean algebra*. The description of other interesting domains may be found in the next section, where specific constraint logic languages are described. For example, CHIP offers finite domains, Trilogy integer arithmetic, Prolog II infinite trees, Prolog III rational numbers, and BNR-Prolog intervals. Other domains such as regular sets and strings are mentioned in the appendix.

2.1 Linear Arithmetic

Arithmetic constraints are maybe the single most important computation domain for constraints. It was also the main motiviation behind the research of combining logic programming with constraints, as standard Prolog handles arithmetics quite poorly. CLP($\mathcal{R}$) [JL87a] was the first constraint logic programming language to introduce linear arithmetic constraints over real or rational numbers. Linear arithmetic constraints correspond to continuous problems, where there is an infinite number of points in the search space to explore.

Linear arithmetic expressions are expressed as terms built from numbers and the operators for change of sign (-), addition (+), subtraction (-), multiplication

(*) and division (/). In order to ensure linearity, one factor of a multiplication and the divisor of a division have to be a number[1] before they can be evaluated. Linear terms can be related to each other using the usual arithmetic constraints ($>, >=, <, =<, =, \neq$).

A suitable *decision procedure* for a set of linear arithmetic constraints is based on the simplex-algorithm (known from linear programming and optimization problems). The procedure either fails if the constraints are not satisfiable or produces a set of bindings for variables and a set of constraints in terms of this variables in a simplified, canonical form.

In important point is that the adaption of the simplex algorithm must be *incremental* to enable a full embedding into a logic language. The reason is that during resolution, new, relatively simple constraints are added (and taken away by backtracking) *dynamically*: If we have already solved a set S of constraints, adding a new constraint C should not require solving the set S $\cup$ {C} from scratch.

The algorithm should also be able to find out if a variable is constraint to exactly one value. If this is the case, the variable can be directly assigned that value. In this way, the inequality constraint ($\neq$) can be handled in a logically correct way. While CHIP [ea88b] claims to support this feature, the current prototype implementation of CLP($\mathcal{R}$) [ea90] available at the CD-Lab does not have this ability. Therefore it does not have logical sound inequality, only a variant definable by the user by negation-as-failure.

Additionally, constraint logic programming languages like CHIP offer extensions to find the most general solution to a set of constraints which optimizes (i.e. minimizes or maximizes) a linear evaluation function.

Non-linear constraints cannot be solved by analytical methods alone in general[2]. Hence iterative methods will be necessary in many cases, which are not implemented in most constraint logic programming languages because of their complexity and numerical instability[3]. The current solution in most systems is to *delay* non-linear expressions until they are bound enough so that they are linear.

This implies that up to now there are no good general suggestions how to handle non-linearity. This includes trigonometric functions as well. It should be noted that there are a number of special purpose systems like Macsyma and Mathematica to perform computer algebra. However, these systems are very complex and it is not clear how they could be integrated into a logic programming environment.

Concluding this subsection, some examples[4] illustrate the power of arithmetic constraints to solve various kinds of problems in scheduling and planning.

[1] Or to be guaranteed to be bound to a number at run-time

[2] But see the language CAL, which is based on Groebner bases

[3] But see BNR-Prolog

[4] "(CLP(R))" in the first line of the programm code indicates an example adopted from the prototype CLP($\mathcal{R}$) [ea90] implementation demo file

The first example defines the well-known *Fibonacci-numbers*. Note that in contrast to standard logic programming, arithmetic expressions can be passed as arguments, as the standard unification is extended to deal with arithmetic constraints.

```
% Fibonacci numbers

fib(0, 1).
fib(1, 1).
fib(N, X1 + X2) :-
    N > 1, X1+X2 > 1,
    fib(N - 1, X1),
    fib(N - 2, X2).

% Sample goals: fib(14,N), fib(X,610)
```

The next example describes the standard mortgage relationship between

- P: Principal
- T: Life of loan in months
- I: Fixed (but compounded) monthly interest rate
- B: Outstanding balance at the end
- M: Monthly payment

Note that although non-linear arithmetic is involved, the sample queries are instantiated sufficiently to produce linear constraints at run-time.

```
% Standard mortgage (CLP(R)):

mg(P, T, I, B, MP) :-
   T = 1,
   B = P + (P*I - MP).
mg(P, T, I, B, MP) :-
   T > 1,
   mg(P*(1 + I) - MP, T - 1, I, B, MP).

% Sample goals: mg(9999, 360, 0.01, 0, M), mg(P, 720, 0.01, B, M)
```

The following example proves that the midpoints of an arbitrary quadrangle form a paralellogram when connected by showing that no constraints hold on the corner points.

```
% Analytical geometry

% A point has two coordinates x and y, written x#y
?- op(31,xfx,#).

mid(AX#AY,BX#BY,CX#CY):-      % compute mid-point of a line
    AX+CX = 2*BX,
    AY+CY = 2*BY.

para(AX#AY,BX#BY,CX#CY,DX#DY):-          % two paralell lines
    (AX-BX)*(CY-DY) = (AY-BY)*(CX-DX).

goal(P0,P1,P2,P3,[P4,P5,P6,P7]):-        % prove it
    mid(P0,P4,P1),
    mid(P1,P5,P2),
    mid(P2,P6,P3),
    mid(P3,P7,P0),
    para(P4,P5,P7,P6),
    para(P4,P7,P5,P6).
```

2.2 Boolean Algebra

Boolean constraint solvers were added to constraint logic programming languages such as CHIP [ea88b], Prolog III [A90a], and CAL [KA88], which already dealt with numerical constraints. *Boolean algebra* is an interesting domain in applications like *circuit design* (development and verification) as well as *theorem-proving* in the domain of propositional calculus. The latter can be applied in expert systems, whose rule yield boolean logic results.

Since boolean unification provides a decision-procedure for propositional calculus and is therefore NP-complete, any algorithm for boolean constraints has an exponential worst case complexity. It is thus very important to use a compact description of boolean terms to achieve efficiency. Normal forms like DNF or sum-of-products require exponential space for the representation of many interesting functions.

CHIP [ea88b], for example, represents boolean terms as directed acyclic graphs, which are manipulated by a special purpose graph manipulation algorithm to eliminate variables. In most cases, however, boolean algebra is implemented as a special case of numerical constraint solving (i.e. the simplex algorithm). In Prolog III [A87] a saturation method is used to solve boolean equations. This method does not compute a most general solution and is therefore not applicable to circuit verification.

Boolean terms are built from *truth values* (*true* and $false$, represented sometimes also by 0 and 1), from *variables* and from *logical connectives* (e.g. and, or, xor, nand, nor, not, =). In some implementations (e.g. CHIP) constants are

also allowed, which denote symbolic names for input arguments.

The following examples illustrates how boolean algebra can be expressed in terms of arithmetic constraints[5]. It also shows the most-cited example in the literature, the full-adder circuit.

```
% Boolean algebra as arithmetic constraints

and(X,Y,Z):- Z = X*Y.

or(X,Y,Z):- Z = X+Y - X*Y.

xor(X,Y,Z):- Z= X+Y - 2*X*Y.

% famous full-adder circuit example

add(I1,I2,I3,O1,O2):-
   xor(I1,I2,X1),
   and(I1,I2,A1),
   xor(X1,I3,O1),
   and(I3,X1,A2),
   or(A1,A2,O2).
```

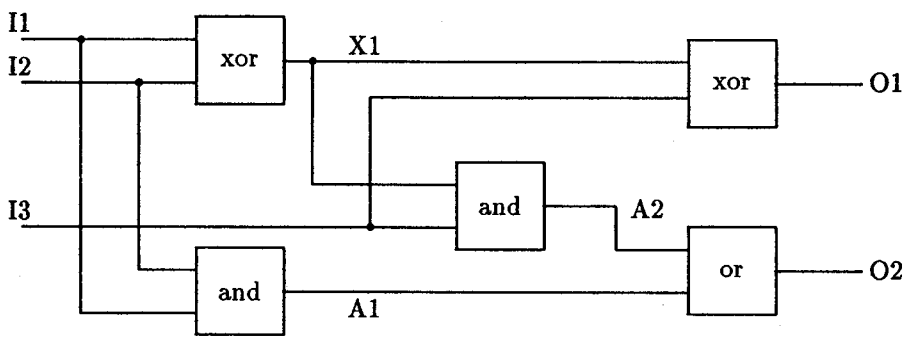

Figure 1: Full Adder Circuit

[5]However not linear ones

3 Current Constraint Logic Programming Languages

Several constraint logic programming languages have now been implemented. Most of these systems include incremental constraint solvers, since constraints are added and deleted dynamically during program execution. CLP($\mathcal{R}$) [JL87a], for example, includes an incremental Simplex algorithm, while CHIP [ea88b, P89] includes an incremental solver for constraints over finite domains.

3.1 The CLP-Scheme (Constraint Logic Programming)

(IBM T.J. Watson Research Center, Yorktown USA)

References: [JS87, JJP87, C87a, L90, JL87a, JL86, JL87b, JJT89, JK88]

Jaffar and Lassez describe a scheme CLP($\mathcal{D}$) for Constraint Logic Programming, which is parameterized by $\mathcal{D}$, the domain of the constraints. In place of substitutions generated by unification, constraints are accumulated and tested for satisfiability over $\mathcal{D}$, using constraint solving techniques appropriate to the domain.

Although languages like Prolog II and Prolog III [A87, A90a] have been proven to be instances of the CLP-scheme, it has certain limitations to be applicable as a general framework for constraint logic programming languages, as remarked in the Ph.D. thesis of Smolka [G89]:

- CLP requires that the constraint language is interpreted in a single fixed domain. For the purpose of knowledge representation, one has to generalize CLP such that the constraint language can come with more than one interpretation to express partial knowledge of the real world.
- CLP requires the interpretations of constraint languages to be *solution compact*, which implies that every element of an interpretation must be obtainable as the unique solution of a possibly infinite set of constraints. CLP needs solution compactness since it provides soundness and completeness results for negation-as-failure. However, the constraint language itself could provide for logical negation.
- CLP assumes that the constraint language is expressed in predicate logic. It is lacking a sufficiently abstract formalization of the notion of a constraint language to accomodate other logics and their customized model theories.

There is a vast literature on theoretic issues of the CLP-scheme. Some practical topics are covered in [JS87, JJP87, C87b, TC88, ea90], where CLP($\mathcal{R}$), a particular instance of the CLP-Scheme over *real numbers* is introduced. Other

authors have proposed other instances of the CLP-scheme, e.g. regular sets [C89] and extensions for knowledge representation [HF88].

The CLP($\mathcal{R}$) system is an interpreter written in about 13000 lines of C-code. The primary aim of its design and construction was to give evidence to the practical potential of the CLP-scheme. The CLP($\mathcal{R}$) system is organized in 3 main parts:

- an *inference engine* which executes derivation steps and maintains variable bindings
- an *interface* which evaluates complex arithmetic expressions and transforms constraints to a canonical form
- a *constraint solver* which solves constraints that are too complicated to be handled directly in the engine and interface, and which also maintains delayed (non-linear) constraints.

A *prototype implementation* of CLP($\mathcal{R}$) is available at the CD-Lab. It is considerably fast. Experiments also indicated that CLP($\mathcal{R}$) is useful for implementing aspects of time-interval based logics with inequalities. CLP($\mathcal{R}$) does a global consistency check, so contradiction and redundancy (by adding the negated constraint and checking for contradiction) can be easily detected. However, CLP($\mathcal{R}$) does not simplify inequalities, for example from the query $X =< 6, 6 =< X$ it does not deduce that $X = 6$[6]. Instead, a canonical form of the above inequalities is returned.

3.2 CHIP (Constraint Handling in Prolog)

(ECRC, Munich Germany)

References: [P89, P87, ea89, PM87, DMP88b, DMP88a, ea88b, T87]

CHIP offers three computation domains for constraints over

- Finite domain restrictive terms
- Boolean terms
- Linear arithmetic terms based on *rational* numbers

There is an ongoing discussion about using either *real* or *rational number arithmetic*. The first approach enables one to solve non-linear expressions like $X * X = 2$, the second approach allows for arbitrary precision and therefore does not have problems with rounding errors, which may invalidate numerical computations with real numbers.

[6]However, it does if the query is written as $(X = 6; X < 6), (X > 6; X = 6)$

The basic feature of CHIP, which distinguishes it from other constraint logic languages, is the ability to work on domain-variables, i.e. variables ranging over a finite domain. CHIP differentiates between two kinds of such variables, those ranging over constants, and those ranging over a finite set of natural numbers. CHIP has also the ability to cope with arithmetic terms over domain-variables, provided their domain are natural numbers.

Finite domains enable a large variety of constraints on domain variables:

- arithmetic constraints, e.g. $>, <, =$
- symbolic constraints, e.g. `element(Nth,Lst,Var)`, `alldistinct(Lst)`
- user-defined constraints using consistency techniques

The following example illustrates an implementation of the classic crypto-arithmetic puzzle. Although this problem could be solved with arithmetic constraints alone as well, the finite domain approach is more efficient. In the example the domain of the variables are the numbers from 0 to 9.

```
  S E N D
+ M O R E
---------
M O N E Y
```

Here it is

```
% The classic puzzle

solve(Digits) :-
      Digits = [S, E, N, D, M, O, R, Y],
      constraints(Digits),
      all_different(Digits),
      Numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
      gen_digits(Digits,Numbers).

constraints([S, E, N, D, M, O, R, Y]) :-
  S >= 0, E >= 0, N >= 0, D >= 0, M >= 0, O >= 0, R >= 0, Y >= 0,
  S <= 9, E <= 9, N <= 9, D <= 9, M <= 9, O <= 9, R <= 9, Y <= 9,
  S >= 1, M >= 1,
  (C1 = 0 ; C1 = 1), (C2 = 0 ; C2 = 1),
  (C3 = 0 ; C3 = 1), (C4 = 0 ; C4 = 1),
  C1          = M,
  C2 + S + M = O + 10 * C1,
  C3 + E + O = N + 10 * C2,
  C4 + N + R = E + 10 * C3,
       D + E = Y + 10 * C4.
```

```
gen_digits([], _).
gen_digits([H | T], L) :-
  element(_, L, H), gen_digits(T, L).
```

At the moment, CHIP is the only language allowing *user-defined constraints* over finite domains. These are solved using so-called *consistency techniques*, a powerful paradigm emerging from AI to solve discrete combinatorical problems. The principle behind these techniques is to use constraints to reduce the domains of variables and thus the size of the search space. This is achieved by propagating the constraints as far as possible and then choosing the most restrictive constraint repeatedly.

There is quite an amount of literature on applications of CHIP claiming ease of implementation and practical speed. However, the drawback of CHIP is that in order to gain efficiency, time-consuming experiments with suitable domain and forward checking declaration, heuristic search rules and the similar are necessary.

3.3 Charme

(Bull CEDIAG, France)

References: [A89]

Charme is maybe the first commercially available constraint logic programming language. It is based on CHIP [ea88b], but has various extensions such as *procedural constructs* like for- and while-loops, which make it look a lot more like a imperative language. In addition, new data-structures like arrays have been added. Also, the syntax has been changed completely, which makes the relationship to logic programming even less clear. A non-trivial application for the car-manufacturer Renault is claimed to have been implemented successfully.

To illustrate the above remarks, a typical predicate definition in Charme might include statements like the following:

```
p(X)
{X=1 or X=2;
withlocal [X,Y] do {X=2;Y=3};
when known(X) do write(X);
 for [X in 1..10, Y in [U,V,W]] do Y!= 2*X;
while extract(Var, Array) do Var > 5}
```

3.4 Prolog II, Prolog III

(Comerauer, Marseille France)

References: [A88, A87, J88, A90a]

Prolog II employs as constraint language equations and disequations that are interpreted in the algebra of *infinite trees*. In this way, Prolog II overcomes the *occur-check problem* by making it a feature. Prolog III adds *rational number arithmetic* and allows for linear equations and inequations for numbers and *boolean expressions* for truth values solved by a saturation method. The semantics of Prolog II and Prolog III are defined by *rewrite rules* over complex trees, not in terms of logical model theory.

Prolog III it is possible to solve finite system of constraints over different domains. The constraints-resolution algorithm replaces the unification algorithm of standard Prolog. For example, to find out the number of pigeons (p) and rabbits (r) required to have a total of 12 heads and 34 legs, one may pose the query

```
{p>=0, r>=0, p+r=12, 2p+4r=34} ?
```

and get the answer

```
p=7, r=5.
```

To compute a list of 9 elements that will produce the same result no matter `<1,2,3>` is concatenated to its left or to its right, the query is

```
{z:9,<1,2,3>*z=z*<1,2,3>} ?
```

The answer is

```
z=<1,2,3,1,2,3,1,2,3>.
```

Note that Prolog III follows a *different syntax* than standard Prolog, called the Marseille syntax, while standard Prolog uses the so-called Edingburgh syntax. In Marseille syntax, variables are written lowercase and lists use angle brackets, logical connectives use different symbols. These syntaxes date back to the first implementations of Prolog in Marseille and Edinburgh respectively.

3.5 CAL (Contrante Avec Logique)

(ICOT, Tokyo Japan)

References: [KA88]

CAL provides an interesting development. Since the CAL interpreter employs the Buchberger algorithm to compute *Groebner bases* of equations as its constraint solver, it can handle *non-linear polynomials* as well as linear ones. A modified version of the algorithm for Groebner bases is used to solve boolean constraints.

The CAL interpreter regards a Groebner base of a system of polynomials as its solution. Given a set of polynomials in homogenous form, then, by the Hilbert zero point theorem, every solution of the set is the solution of some polynomial if some power of the polynomial is in the *ideal*. Consequently, the set of polynomials does not have a solution if and only if 1 is in the generated ideal. Now the problem of solving contraints is reduced to the membership problem of the generated ideal. Buchberger gave an algorithm for this problem. Each equation is viewed as a *rewrite rule* which rewrites the maximum monomial to the rest of the polynomial under a certain ordering between monomials. *Critical pairs* are handled until a confluent rewriting system is resulting from the algorithm, which is called the Groebner base of the initial set of polynomials. Now a polynomial is contained in an ideal if and only if the polynomial is reduced to zero by the rewriting under its Groebner base.

One problem of the approach is that that the computation of the Groebner bases is usually *exponential* in the size of the polynomials and that the general algorithm has to be adopted carefully to be incremental.

3.6 Trilogy

(University of Vancouver, Vancouver Canada)

References: [P88]

Unlike CLP($\mathcal{R}$) [JL87a], Prolog III [A90a] and CHIP [ea88b], the language Trilogy does not provide the full power of Prolog. It is rather a *hybrid* with the more conventional language Pascal. From the constraint point of view, it provides a decision procedure for *Pressburger arithmetic*, which is arithmetic on linear expressions (including the modulo-operator) over natural numbers. Trilogy is available for around 100$ for IBM-PC or compatibles.

3.7 BNR-Prolog (Bell-Northern Research Prolog)

(Bell-Northern Research, Ottawa Canada)

References: [OWB89, G88]

The main feature of BNR-Prolog is an implementation of *interval arithmetic.* Constraint arithmetic on intervals restores not only a declarative reading to arithmetic expression but also their algebraic properties. Thus, while floating point rounding errors will typically cause the functional evaluation of equaltiy expressions like

$$(x + y) + z =:= x + (y + z)$$

to fail, such arithmetic operations on interval values of the variables cannot contain rounding errors and they are guaranteed to succeed.

When combined with ordinary backtracking of Prolog, relational interval arithmetic can also be used to obtain numeric solutions to *non-linear* constraint satisfaction problems over the reals (e.g. like n-degree polynomials).

This technique differs from other approaches like Prolog III [A87, A90a] or CLP($\mathcal{R}$) [JL87a] in that it does not do any term-rewriting or equation solving. In interval arithmetic, intervals are narrowed by raising their lower bounds or lowering their upper bounds. For example, assume that $X = [Xlb, Xub]$ and $Y = [Ylb, Yub]$ are constrained by equality, then both X and Y are narrowed to the interval $[max(Xlb, Ylb), min(Xub, Yub)]$. Another example shows that the evaluation of a relational equation may well narrow all the intervals in it. Given

$$X + Y =:= Z \texttt{ with } X = [3,7], Y = [2,8], Z = [4,6]$$

the variables are narrowed to the intervals:

$$X = [3,4], Y = [2,3], Z = [5,6]$$

The advantage of interval arithmetic is that it can deal with non-linear and trigonometric arithmetic expressions, the disadvantage is that interval arithmetic cannot solve even simple sets of linear equations and that the narrowing process is sometimes inefficient or may not even converge.

3.8 CS-Prolog (Constraint Solver Prolog)

(University of Tokyo, Tokyo Japan)

References: [ea87]

CS-Prolog implements some basic constraint solving techniques in Prolog itself. Namely, an equation solver based on variable elimination and term rewriting, an implementation of finite domains (see CHIP [ea88b]), and an inequation solver based on a graph search technique are presented.

3.9 CIL (Complex Indeterminates Language)

(Mukai, Japan)

References: [KH85]

CIL is a logic language for natural language understanding based on situation semantics. CIL is a *knowledge representation language* assuming that knowledge is represented by parametrized types and constraints between them and that constraints are described by Horn clauses. CIL = horn clause logic + types and complex indeterminates + delay mechanism (freeze).

Although CIL can be viewed as constraint language, it does not cover arithmetic constraints, which are the main area of interest of this report. For more on this topic, compare CIL to proposals like [HR86, HF88].

3.10 The cc-Scheme (Concurrent Constraints)

(Saraswat, Stanford University)

References: [A90b]

Saraswat's Ph.D. dissertation describes a family of *concurrent constraint languages*. It is based on the notion of partial information, and the concommitant notions of consistency and entailment. The family is founded on the CLP-scheme on one hand and concurrent logic programming on the other hand. In this framework, computation emerges from the interaction of concurrently executing agents that communicate by placing, checking and instantiating constraints on shared variables. The *state* of a concurrent system is specified by a store, which is a vector of variables, and a valuation assigning each variable a completely known value in its domain. Then a *constraint* is defined as a set of such valuations. The *store* can be read and written enabling a transformation of states.

This short description should suffice to indicate that Saraswats view of concurrent constraint programming is highly abstract and at the moment probably more interesting from the concurrent programming point of view than from the constraint programming point of view. Currently it is not clear how to implement and utilize such languages. In [GHE89] semantics for the Ask-and-Tell class of constraint-based concurrent logic programming languages are given based upon the notion of *reactive behaviors.*

4 Conclusions

Hopefully this report could give a first idea about Constraint Logic Programming, which extends usual Prolog-like logic programming languages by introduc-

ing constraints on specific computation domains. Keeping the declarativeness and *flexibility* of fifth-generation tools, constraints bring into logic programming the *efficiency* of special purpose programs written in traditional imperative languages.

This very active area of research promises some very interesting possibilities for real life applications, which are often combinatorical explosive but can be easily formulated with the help of constraints. Tasks like *scheduling, planning* and *circuit design* could greatly benefit from these developments.

At the current stage of development, practically all constraint logic programming languages except the recently introduced Prolog III implementation [A90a] are not polished commercial products, but rather academic prototype versions. Among these, **the CD-Lab offers CLP($\mathcal{R}$) and Prolog II.**

For the interested reader, it should be noted that a survey on the same topic was published in the ACM Communications [J90] just after this report was finished.

References

[A87] Colmerauer A. Opening the Prolog-III Universe. Byte Magazine, Special Issue on Logic Programming, August 1987.

[A88] Colmerauer A. Solving Equations and Inequations on Finite and Infinite Trees. Fifth Generation Computing Systems Conference, Tokyo Japan, December 1988.

[A89] Oplobedu A. Charme: Un Langage Industriel de Programmation par Contraintes. Expert Systems Conference, Avignon France, 1989.

[A90a] Colmerauer A. An Introduction to Prolog III. ACM Communications Vol 33, Number 7, July 1990.

[A90b] Saraswat V A. Concurrent Constraint Programming. 17th ACM Symp on Principles of Programming Languages, San Francisco USA, January 1990.

[C87a] Lassez C. Constraint Logic Programming. Byte Magazine, Special Issue on Logic Programming, August 1987.

[C87b] Lassez C. Constraint Logic Programming and Option Trading. IEEE Expert, Special fall issue on financial software, August 1987.

[C89] Walinsky C. Constraint Logic Programming with Regular Sets. Technical Report, Dartmouth College, Hanover USA, 1989.

[DMP88a] Simonis H Dincbas M and Van Hentenryck P. Solving a Cutting-Stock Problem in Constraint Logic Programming. 5th Int Conf and Symp on Logic Programming, Seattle USA, August 1988.

[DMP88b] Simonis H Dincbas M and Van Hentenryck P. Solving the Car-Sequencing Problem in Constraint Logic Programming. 8th European Conference on Artificial Intelligence, August 1988.

[ea87] Kawamura T et al. CS-Prolog: A Generalized Unification Based Constraint Solver. 6th Int Conf on Logic Programming, Tokyo Japan, June 1987.

[ea88a] Borning A et al. Constraint Hierarchies and Logic Programming. Technical Report 88-11-10, University of Washington, USA, November 1988.

[ea88b] Dincbas M et al. The Constraint Logic Programming Language Chip. Fifth Generation Computing Systems Conference, Tokyo Japan, December 1988.

[ea89] Graf T et al. Simulation of Hybrid Circuits in Constraint Logic Programming. 11th Int Joint Conf on Artificial Intelligence 89, August 1989.

[ea90] Jaffar J et al. CLP($\mathcal{R}$) Version 1.0 - Reference Manual. Technical Report, IBM T.J. Watson Research Center, Yorktown USA, May 1990. Draft.

[FBBNA90] Maloney J Freeman-Benson B N and Borning A. An Incremental Constraint Solver. ACM Communications Vol 33, Number 1, January 1990.

[G88] Cleary J G. Logical Arithmetic. Future Computing Systems, 2(2) 1987, Oxford University Press, 1988.

[G89] Smolka G. Logic Programming over Polymorphically Order-Sorted Types. Ph.D. Thesis, Universitaet Kaiserslautern, Germany, May 1989.

[GHE89] Maher M J Gaifman H and Shapiro E. Reactive Behaviour Semantics for Concurrent Constraint Logic Programs. Technical Report CS89-22, Weizmann Institute of Science, Rehovot Israel, September 1989.

[HF88] Beringer H and Porcher F. A Relevant Scheme for Prolog Extensions: CLP(Conceptual Theory). Technical Report, Electronique Serge Dassault, Saint Cloud France, 1988.

[HR86] Ait-Kaci H and Nasr R. Login: A Logic Programming Language with Built-In Inheritance. Journal of Logic Programming, 3(3), 1986.

[J88] Maher M J. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. 3rd Symp on Logic in Computer Science, Edinburgh Scotland, July 1988.

[J90] Cohen J. Constraint Logic Programming Languages. ACM Communications Vol 33, Number 7, July 1990.

[JJP87] Michaylov S Jaffar J and Stuckey P. CLP($\mathcal{R}$) and Some Electrical Engineering Problems. 4th Int Conf on Logic Programming 87, Melbourne, May 1987.

[JJT89] McAloon K Jaffar J and Huynh T. Simplification and Elimination of Redundant Linear Arithemtic Constraints. North American Conf on Logic Programming 89, Cleveland USA, October 1989.

[JK88] Jaffar J and McAloon K. Applications of a Canonical Form for Generalized linear Constraints. Fifth Generation Computing Systems Conference, Tokyo Japan, December 1988.

[JL86] Jaffar J and Lassez J L. Constraint Logic Programming (Theory). Technical Report, Department of Computer Science, Monash University, Australia, June 1986. Draft.

[JL87a] Jaffar J and Lassez J L. Constraint Logic Programming. 14th ACM Symp on Principles of Programming Languages, Munich, January 1987.

[JL87b] Jaffar J and Lassez J L. From Unification to Constraints. 6th Int Conf on Logic Programming, Tokyo Japan, June 1987.

[JS87] Jaffar J and Michaylov S. Methodology and Implementation of a CLP system. 4th Int Conf on Logic Programming 87, Melbourne, May 1987.

[KA88] Sakai K and Aiba A. Introduction to CAL (Extended Abstract). ICOT Memo, January 1988.

[KH85] Mukai K and Yasukawa H. Complex Indeterminates in Prolog and its Application to Discourse Models. New Generation Computing 3, 1985.

[L80] Steele G L. The Definition and Implementation of a Computer Programming Language based on Constraints. Ph.D. Thesis, MIT AI-TR 595, Massachusetts USA, August 1980.

[L90] Lassez J L. Parametric Queries, Linear Constraints and Variable Elimination. Design and Implementation of Computation Systems (DISCO 90), Springer LNCS 429, April 1990.

[OWB89] Vellino A Older W and Farrahi B. Relational Arithmetic in Prolog using Interval Constraints. Technical Report, Bell-Northern Computing Research Laboratory, Ontario Canada, 1989.

[P87] Van Hentenryck P. A Theoretical Framework for Consistency Techniques in Logic Programming. Int Joint Conf on Artificial Intelligence 87, Milan Italy, August 1987.

[P88] Voda P. The Constraint Language Trilogy: Semantics and Computations. Technical Report CLS, Vancouver Canada, 1988.

[P89] Van Hentenryck P. Constraint Satisfaction in Logic Programming. MIT Press, Cambridge, MA, 1989.

[PM87] Van Hentenryck P and Dincbas M. Forward Checking in Logic Programming. 4th Int Conf on Logic Programming 87, Melbourne, May 1987.

[T87] Graf T. Extending Constraint Handling in Logic Programming to Rational Arithmetic. Master Thesis, Technical University Vienna, Austria, September 1987. Draft.

[TC88] Huynh T and Lassez C. A CLP($\mathcal{R}$) Options Trading Analysis System. 5th Int Conf and Symp on Logic Programming, Seattle USA, August 1988.

[Wm88] Leler Wm. Constraint Programming Languages: Their Specification and Generation. Addison-Wesley Publishing Comp, New York USA, 1988.

Contents