A Proof Theoretic View of Constraint Programming

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dept. of Mathematics, Computer Science, Physics & Astronomy
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

January 14, 1998

Abstract

We provide here a proof theoretic account of constraint programming that attempts to capture the essential ingredients of this programming style. We exemplify it by presenting proof rules for linear constraints over interval domains, and illustrate their use by analyzing the constraint propagation process for the `SEND + MORE = MONEY` puzzle. We also show how this approach allows one to build new constraint solvers.

1 Introduction

1.1 Motivation

One of the most interesting recent developments in the area of programming has been constraint programming. A prominent instance of it is *constraint logic programming* exemplified by such programming languages as CLP($\mathcal{R}$), Prolog III or ECL^{i}PS^e. But recently also imperative constraint programming languages emerged, such as 2LP of McAloon & Tretkoff (1995) or CLAIRE of Caseau & Laburthe (1996). (For an overview of this area and related references see Hentenryck, Saraswat & et al. (1996)).

The aim of this paper is to explain the essence of this approach to programming without committing oneself to a particular programming paradigm. We achieve this by providing a simple proof theoretic framework that allows us in particular to explain *constraint propagation*, one of the cornerstones of constraint programming.

The simplicity and elegance of constraint logic programming has already led to a general presentation of their operational semantics in Jaffar & Maher (1994) that easily can be casted in a proof theoretic jacket. But this account is limited to the logic programming view of constraint programming. Moreover, it treats constraint propagation as a further unexplained atomic action. Admittedly, the latter deficiency has been addressed in van Emden (1997), where it has been explained how constraint propagation can be defined within the framework of Jaffar & Maher (1994).

In our approach we try to "decouple" constraint programming from logic programming by going back to the origins of constraints handling and by viewing computing as a task of transforming one *constraint satisfaction problem* (CSP) into another, equivalent one. To take into account reasoning by case analysis (that leads to "don't-know" nondeterminism and various forms of backtracking) we further introduce a splitting operation that allows us to split one CSP into two, the union of which is equivalent to the original CSP. The rules that govern this process of transforming one CSP into a finite collection of them fall naturally into four categories and seem to be sufficient to describe the computation process.

By providing such a general view of constraint programming we can use it to analyze the constraint programming process both within the logic programming paradigm and the imperative one. In particular, as we shall see in Sections 3 and 4, we can use it both to study existing constraint solvers and to build new ones.

As a by-product of these considerations we bring constraint programming closer to the *computation as deduction* paradigm according to which the computation process is identified with a constructive proof of a formula (a query) from a set of axioms. This paradigm goes back to Herbrand and Gödel and is exemplified by logic programming and functional programming and also by viewing the parsing process as a deduction (see e.g., Sieber, Schabes & Pereira (1995)).

1.2 Related Work

A number of papers have advocated theorem proving as a means to account for various aspects constraint logic programming. In particular, a proof theoretic approach to constraint propagation within the constraint logic programming framework has been proposed in Frühwirth (1995). In this work so-called constraint handling rules (CHR) have been introduced. CHR are available as part of the ECL^{i}PS^e system and allow the user to define his/her own constraint solvers.

Next, in Andrews (1995) proof-theoretic techniques are used to compare the intended theory and its actual implementation for various constraint logic programming systems.

Further, a related to ours approach to constraint programming has been proposed in Wetzel, Kowalski & Toni (1995). In this work constraint logic programs are identified with so-called if-and-only-if definitions augmented with integrity constraints. The if-and-only-if definitions are used to define the "usual" logic programming computation step while the integrity constraints are used to account for the constraint propagation process that is identified with a complete resolution strategy.

Our view of constraint programming is also compatible with that expounded in Smolka (1996) where constraint programming is presented without a commitment to a specific programming paradigm. In fact, our approach allows one to couch his concepts of propagators, inference engines and distributors into a more specific, proof theoretic, framework.

The approach here presented is closest to the one introduced independently in Castro (1997). Even though the overall objectives are essentially the same, the emphasis in his paper lies rather on defining specific techniques of constraint programming such as arc consistency and forward checking by means of proof rules and strategies.

1.3 Preliminaries

Consider a finite sequence of variables $\mathcal{X} := x_1, \ldots, x_n$ where $n \geq 0$, with respective domains $\mathcal{D} := D_1, \ldots, D_n$ associated with them. So each variable x_i ranges over the domain D_i. By a *constraint* C on $\mathcal{X}$ we mean a subset of $D_1 \times \ldots \times D_n$. If C equals $D_1 \times \ldots \times D_n$ then we say that C is *solved*. In the boundary case when $n = 0$ we admit two constraints, denoted by $\top$ and $\perp$, that denote respectively the *true constraint* (for example $0 = 0$) and the *false constraint* (for example $0 = 1$).

By a *constraint satisfaction problem*, *CSP* in short, we mean a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{C}$ is a finite set of constraints, each on a subsequence of $\mathcal{X}$.

We say that an n-tuple $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ *is a solution to* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ if for every constraint $C \in \mathcal{C}$ on the variables $x_{i_1}, \ldots, x_{i_m}$ from $\mathcal{X}$ we have

$$(d_{i_1}, \ldots, d_{i_m}) \in C.$$

Below we represent a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ as an expression of the form $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$, where $\mathcal{DE} := \{x_1 \in D_1, \ldots, x_n \in D_n\}$. We call a construct of the form $x \in D$ a *domain expression.* To simplify the notation from now on we omit the "{ }" brackets when presenting $\mathcal{C}$ and $\mathcal{DE}$.

We call a CSP *solved* if it is of the form $\langle \emptyset \ ; \ \mathcal{DE} \rangle$ where no domain in $\mathcal{DE}$ is empty, and *failed* if it either contains the false constraint $\perp$ or some of its domains is empty. So a failed CSP admits no solution.

Given two CSP's ϕ and ψ, we call ϕ a *variant* of ψ if the removal of solved constraints from ϕ and ψ yields the same CSP.

In what follows we assume that the constraints and the domain expressions are defined in some specific, further unspecified, language. In this representation of the constraints it is implicit that each of them is a subset of the Cartesian product of the associated variable domains. For example, if we consider the CSP $\langle x < y \ ; \ x \in [0..10], y \in [5..10] \rangle$, then we view the constraint $x < y$ as the set $\{(a, b) \mid a \in [0..10], b \in [5..10], a < b\}$.

Given a constraint c on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, and a sequence of domains $D'_1, \ldots, D'_n$ such that for $i \in [1..n]$ we have $D'_i \subseteq D_i$, we say that c' is the *result of restricting* c *to the domains* $D'_1, \ldots, D'_n$ if $c' = c \cap (D'_1 \times \ldots \times D'_n)$.

2 The Proof Theoretic Framework

In this section we introduce a proof theoretic framework that will be used throughout the paper.

2.1 Format of the Proof Rules

In what follows we consider two types of proof rules that we call *deterministic* and *splitting.* The deterministic rules are of the form

$$\frac{\phi}{\psi}$$

where ϕ and ψ are CSP's. We assume here that ϕ is not failed and its set of constraints is non-empty. Depending on the form of the conclusion ψ we distinguish two cases.

- *Domain reduction rules*, or in short *reduction rules.* These are rules in which the domains of the conclusion are respective subsets of the domains of the premise and the constraints of the conclusion are respective restrictions of the constraints of the premise to the domains of the conclusion. So they have the following form:

  $$\frac{\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \mathcal{C}' \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

  where for $i \in [1..n]$ we have $D'_i \subseteq D_i$ and where $\mathcal{C}'$ is the result of restricting each constraint in $\mathcal{C}$ to the corresponding subsequence of the domains $D'_1, \ldots, D'_n$.

  In an extremal case when a domain of one or more variables gets reduced to the empty set, the resulting CSP becomes failed.

  When all constraints in $\mathcal{C}'$ are solved, we call such a rule a *solving rule.*

- *Transformation rules.* These rules are not domain reduction rules and have the following form:

$$\frac{\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle}{\langle \mathcal{C}' \ ; \ \mathcal{DE}' \rangle}$$

  where $\mathcal{C}' \neq \emptyset$ and $\mathcal{DE} \subseteq \mathcal{DE}'$. The inclusion between $\mathcal{DE}$ and $\mathcal{DE}'$ takes care of the possibility that constrains on some new variables have been introduced in $\mathcal{C}'$ — for these variables new domain expressions have to be added.

  Here a failure is reached when the false constraint is generated.

The splitting rules are of the form

$$\frac{\phi}{\psi_1 \mid \psi_2}$$

where ϕ, ψ_1 and ψ_2 are CSP's. As for deterministic rules we assume here that ϕ is not failed and its set of constraints is non-empty. In what follows we only consider splitting rules in which ϕ, ψ_1 and ψ_2 are CSP's with the same sequence of variables.

These rules allow us to replace one CSP by two CSP's. The intuition is that their "union" is "equivalent" to the original CSP. They are counterparts of the rules just introduced. So, again, we distinguish two cases.

- *Reduction splitting rules.* These are rules such that both $\frac{\psi}{\phi_1}$ and $\frac{\psi}{\phi_2}$ are reduction rules.

- *Transformation splitting rules.* These are rules such that both $\frac{\psi}{\phi_1}$ and $\frac{\psi}{\phi_2}$ are transformation rules.

2.2 Examples of Proof Rules

In the sequel when presenting specific proof rules we delete from the conclusion all solved constraints and all domain expressions that do not change. Also, we abbreviate the domain expression $x \in \{a\}$ to $x = a$.

As an example of a reduction rule consider the following rule:

EQUALITY 1

$$\frac{\langle x = y \ ; \ x \in D_1, y \in D_2 \rangle}{\langle x = y \ ; \ x \in D_1 \cap D_2, y \in D_1 \cap D_2 \rangle}$$

Note that this rule yields a failure when $D_1 \cap D_2 = \emptyset$. In case $D_1 \cap D_2$ is a singleton this rule becomes a solving rule, that is the constraint $x = y$ becomes solved (and hence deleted). Note also the following solving rule:

EQUALITY 2

$$\frac{\langle x = x \ ; \ x \in D \rangle}{\langle \ ; \ \rangle}$$

Following the just introduced convention we dropped the constraint and the domain expression from the conclusion of the *EQUALITY 2* rule. This explains its degenerated format.

As further examples of solving rules consider the following three concerning disequality:

DISEQUALITY 1

$$\frac{\langle x \neq x \ ; \ x \in D\rangle}{\langle \ ; \ x \in \emptyset\rangle}$$

DISEQUALITY 2

$$\frac{\langle x \neq y \ ; \ x \in D_1, y \in D_2\rangle}{\langle \ ; \ \rangle}$$

where $D_1 \cap D_2 = \emptyset$,

DISEQUALITY 3

$$\frac{\langle x \neq y \ ; \ x \in D, y = a\rangle}{\langle \ ; \ x \in D - \{a\}\rangle}$$

where $a \in D$, and similarly with $x \neq y$ replaced by $y \neq x$.

So the *DISEQUALITY 1* rule yields a failure while the *DISEQUALITY 3* rule can yield a failure.

Next, as an example of a transformation rule consider the following rule that substitutes a variable by a value:

SUBSTITUTION

$$\frac{\langle \mathcal{C} \ ; \ \mathcal{DE}, x = a\rangle}{\langle \mathcal{C}\{x/\overline{a}\} \ ; \ \rangle}$$

where x occurs in $\mathcal{C}$.

Here $\overline{a}$ stands for the constant that denotes in the underlying language the value a and $\mathcal{C}\{x/\overline{a}\}$ denotes the set of constraints obtained from $\mathcal{C}$ by substituting in it every occurrence of x by $\overline{a}$. So x does not occur in $\mathcal{C}\{x/\overline{a}\}$.

Another example of a transformation rule forms the following rule:

DELETION

$$\frac{\langle \mathcal{C} \cup \{\top\} \ ; \ \mathcal{DE}\rangle}{\langle \mathcal{C} \ ; \ \rangle}$$

Let us consider now the splitting rules. A natural class of examples of reduction splitting rules form *domain splitting rules*. These are rules of the form:

$$\frac{\langle \mathcal{C} \ ; \ \mathcal{DE}, x \in D\rangle}{\langle \mathcal{C}' \ ; \ x \in D_1\rangle \mid \langle \mathcal{C}'' \ ; \ x \in D_2\rangle}$$

where $D_1 \cup D_2 = D$, $\mathcal{C}'$ is the result of restricting each constraint in $\mathcal{C}$ to the corresponding subsequence of the domains in $\mathcal{DE}$ and D_1, and analogously with $\mathcal{C}''$.

If such a rule does not depend on $\mathcal{C}$ and $\mathcal{DE}$ (the latter, following the introduced convention, is omitted from the conclusions of the rule) we abbreviate it to

$$\frac{x \in D}{x \in D_1 \mid x \in D_2}$$

Two specific instances are:

$$\frac{\textit{ENUMERATION}}{}$$

$$\frac{x \in D}{x = a \mid x \in D - \{a\}}$$

where D is a finite domain with at least two elements and $a \in D$, and

BISECTION

$$\frac{x \in [a..b]}{x \in [a..\frac{a+b}{2}] \mid x \in [\frac{a+b}{2}..b]}$$

where $[a..b]$ a closed non-empty interval of reals.

Finally, a natural class of examples of transformation splitting rules form *constraint splitting rules.* They have the following form:

$$\frac{\langle \mathcal{C}, C \ ; \ \mathcal{DE} \rangle}{\langle \mathcal{C}, C_1 \ ; \ \rangle \mid \langle \mathcal{C}, C_2 \ ; \ \rangle}$$

where

- C, C_1 and C_2 are constraints on the same sequence of variables,
- every solution to $\langle \mathcal{C}, C \ ; \ \mathcal{DE} \rangle$ is is a solution to $\langle \mathcal{C}, C_1 \ ; \ \mathcal{DE} \rangle$ or $\langle \mathcal{C}, C_2 \ ; \ \mathcal{DE} \rangle$,
- every solution to $\langle \mathcal{C}, C_i \ ; \ \mathcal{DE} \rangle$ $(i \in [1,2])$ is a solution to $\langle \mathcal{C}, C \ ; \ \mathcal{DE} \rangle$.

If such a rule does not depend on $\mathcal{C}$ and $\mathcal{DE}$ we abbreviate it to

$$\frac{C}{C_1 \mid C_2}$$

A particular instance is:

$$\frac{|x - y| = a}{x - y = a \mid x - y = -a}$$

where x and y are integer variables and a is an integer.

2.3 Derivations

Now that we have defined the proof rules, we define the result of applying a proof rule to a CSP. Assume a CSP of the form $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1 \rangle$. First, consider a deterministic rule, so a rule of the form

$$\frac{\langle \mathcal{C}_1 \ ; \ \mathcal{D}_1 \rangle}{\langle \mathcal{C}_2 \ ; \ \mathcal{D}_2 \rangle} \tag{1}$$

Rename the variables of $\langle \mathcal{C}_2 \ ; \ \mathcal{D}_2 \rangle$ that appear in $\mathcal{C}$ but not in $\mathcal{C}_1$ so that they do not appear in $\mathcal{C}$ and denote the so obtained CSP by $\langle \mathcal{C}_2' \ ; \ \mathcal{D}_2' \rangle$.

We then say that the above proof rule *can be applied* to $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1 \rangle$ and call

$$\langle \mathcal{C} \cup \mathcal{C}_2' \ ; \ \mathcal{D} \cup \mathcal{D}_2' \rangle$$

the *result of applying rule* (1) *to* $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1 \rangle$. If $\langle \mathcal{C} \cup \mathcal{C}_2' \ ; \ \mathcal{D} \cup \mathcal{D}_2' \rangle$ is not a variant of $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1 \rangle$, then we say that it is the result of a *relevant application of rule* (1) *to* $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1 \rangle$.

Further, given a CSP ϕ and a deterministic rule R, we say that ϕ is *closed under the applications of* R if either R cannot be applied to ϕ or no application of it to ϕ is relevant.

For example, assuming for a moment the expected interpretation of propositional formulas, the CSP $\phi := \langle x \wedge y = z \ ; \ x = 1, y = 0, z = 0\rangle$ is closed under the applications of the transformation rule

$$\frac{\langle x \wedge y = z \ ; \ x = 1, y \in D_y, z \in D_z\rangle}{\langle z = y \ ; \ \rangle}$$

Indeed, this rule can be applied to ϕ; the outcome is $\psi := \langle z = y \ ; \ x = 1, y = 0, z = 0\rangle$. After the removal of solved constraints from ϕ and ψ we get in both cases the solved CSP $\phi := \langle \emptyset \ ; \ x = 1, y = 0, z = 0\rangle$.

In contrast, the CSP $\phi := \langle x \wedge y = z \ ; \ x = 1, y \in \{0,1\}, z \in \{0,1\}\rangle$ is not closed under the applications of the above rule because $\langle z = y \ ; \ x = 1, y \in \{0,1\}, z \in \{0,1\}\rangle$ is not a variant of ϕ.

Next, consider a splitting rule, so a rule of the form

$$\frac{\langle \mathcal{C}_1 \ ; \ \mathcal{D}_1\rangle}{\langle \mathcal{C}_2 \ ; \ \mathcal{D}_2\rangle \mid \langle \mathcal{C}_3 \ ; \ \mathcal{D}_3\rangle} \tag{2}$$

We then say that rule (2) *can be applied* to $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1\rangle$ and call

$$\langle \mathcal{C} \cup \mathcal{C}_2 \ ; \ \mathcal{D} \cup \mathcal{D}_2\rangle \mid \langle \mathcal{C} \cup \mathcal{C}_3 \ ; \ \mathcal{D} \cup \mathcal{D}_3\rangle \tag{3}$$

the *result of applying it to* $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1\rangle$. If neither $\langle \mathcal{C} \cup \mathcal{C}_2 \ ; \ \mathcal{D} \cup \mathcal{D}_2\rangle$ nor $\langle \mathcal{C} \cup \mathcal{C}_3 \ ; \ \mathcal{D} \cup \mathcal{D}_3\rangle$ is a variant of $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1\rangle$, then we say that (3) is the result of a *relevant application of rule* (2) *to* $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{D} \cup \mathcal{D}_1\rangle$. (Recall that by assumption all three CSP's $\langle \mathcal{C}_i \ ; \ \mathcal{D}_i\rangle$, where $i \in [1..3]$, have the same sequence of variables, so we do not need to worry here about variable clashes.)

Finally, we introduce the notions of a proof tree and of a derivation.

Definition 2.1 Assume a set of proof rules. A *proof tree* is a tree the nodes of which are CSP's. Further, each node has at most two direct descendants and for each node ϕ the following holds:

- If ϕ is a leaf, then no application of a rule to ψ is relevant;
- If ϕ has precisely one direct descendant, say ψ, then ψ is the result of a relevant application of a proof rule to ϕ;
- If ϕ has precisely two direct descendants, say ψ_1 and ψ_2, then $\psi_1 \mid \psi_2$ is the result of a relevant application of a proof rule to ϕ.

A *derivation* is a branch in a proof tree. A derivation is called *successful* if it is finite and its last element is a solved CSP. A derivation is called *failed* if it is finite and its last element is a failed CSP. □

The idea behind the above definition is that we consider in the proof trees only those applications of the proof rules that cause some change. Note also that more proof rules can be applicable to a given CSP, so a specific CSP can be a root of several proof trees.

Note that some finite derivations are neither successful nor failed. In fact, many constraint solvers yield CSP's that are neither solved nor failed — their aim is to bring the initial CSP to some specific, simpler form.

In some cases this third possibility does not arise. Indeed, consider a non-failed CSP with a non-empty set of constraints on finite domains. Then either the *DELETION* or the *SUBSTITUTION* or the *ENUMERATION* rule can be applied to it and moreover each such application is always relevant. So in presence of the above three rules for CSP's with finite domains each finite derivation is either successful or failed.

2.4 Equivalent CSP's

We introduced the proof rules so that we can reduce one CSP to another CSP or to two CSP's, which are in some sense "smaller" yet "equivalent". Both notions can be made precise but the first one will not play any role in our considerations, so we only present an adequate notion of equivalence. Because the considered proof rules are of a specific form, we limit ourselves in the definition to specific pairs of CSP's.

Definition 2.2 Consider two CSP's ϕ and ψ such that all variables of ϕ are also present in ψ. We say that the CSP's ϕ and ψ are *equivalent* if

- every solution to ϕ is or can be extended to a solution to ψ,
- for every solution to ψ its restriction to the variables of ϕ is a solution to ϕ. □

In particular, two CSP's with the same sequence of variables are equivalent if they have the same set of solutions. So for example the CSP's

$$\langle 3x - 5y = 4 \ ; \ x \in [0..9], y \in [1..8] \rangle$$

and

$$\langle 3x - 5y = 4 \ ; \ x \in [3..8], y \in [1..4] \rangle$$

are equivalent, since both them have $x = 3, y = 1$ and $x = 8, y = 4$ as the only solutions, and, for $\mathcal{DE} := x \in D_x, y \in D_y, z \in D_z$, so are

$$\langle x < y, y < z \ ; \ \mathcal{DE} \rangle$$

and

$$\langle x < y, y < z, x < z \ ; \ \mathcal{DE} \rangle.$$

In contrast,

$$\langle x < z \ ; \ x \in D_x, z \in D_z \rangle$$

and

$$\langle x < y, y < z \ ; \ \mathcal{DE} \rangle$$

are not equivalent, as not each solution to the former extends to a solution of the latter.

This brings us to the following notion where we make use of the fact that the considered proof rules are of a specific form.

Definition 2.3

(i) A proof rule

$$\frac{\phi}{\psi}$$

is called *equivalence preserving* if ϕ and ψ are equivalent.

(ii) A proof rule

$$\frac{\phi}{\psi_1 \mid \psi_2}$$

is called *equivalence preserving* if

- every solution to ϕ is a solution to ψ_1 or to ψ_2,
- every solution to ψ_i ($i \in [1,2]$) is a solution to ϕ. □

All the rules discussed so far are equivalence preserving. From the way we introduce the proof rules in the sequel it will be clear that all of them are also equivalence preserving.

This completes the presentation of our proof theoretic framework. In order to use it to model constraint programming the proof rules above introduced have to be "customized" to a specific language in which constraints are defined and to specific domains. In what follows we present an example of such a customization that deals with linear constraints over interval and finite domains. In Apt (1998) the case of Boolean constraints is considered.

Such rules should be selected and scheduled in an appropriate way and some strategy should be employed to traverse the generated proof trees. We defer discussion of these issues to Subsection 5.2.

3 Linear Constraints over Interval Domains

In this section we consider linear constraints over interval domains. We use the introduced rules to discuss the behaviour of the ECL^{i}PS^e finite domain solver and, by means of an example, to analyse the *SEND + MORE = MONEY* puzzle.

First, let us recall the relevant definitions. By a *linear expression* we mean a term in the language that contains two constants 0 and 1, the unary minus function $-$ and two binary functions $+$ and $-$, both written in the infix notation. We abbreviate terms of the form

$$\underbrace{1 + \ldots + 1}_{n \text{ times}}$$

to n, terms of the form

$$\underbrace{x + \ldots + x}_{n \text{ times}}$$

to nx and analogously with -1 and $-x$ used instead of 1 and x. So (using appropriate transformation rules) each linear expression can be equivalently written in the form

$$a_1x_1 + \ldots + a_nx_n + a_{n+1}$$

where $n \geq 0$, $a_1, \ldots, a_n$ are non-zero integers, $x_1, \ldots, x_n$ are different variables and a_{n+1} is an integer.

By a *linear constraint* we mean a formula of the form

$$s \; op \; t$$

where s and t are linear expressions and $op \in \{<, \leq, =, \neq, \geq, >\}$. In what follows we drop the qualification "linear" when discussing linear expressions and linear constraints.

Further, we call

- $s \leq t$ an *inequality constraint*,
- $s = t$ an *equality constraint*,
- $s \neq t$ a *disequality constraint*,

- $x \neq y$, for variables x, y, a *simple disequality constraint.*

By an *interval* we mean an expression of the form

$$[a..b]$$

where a and b are integers; $[a..b]$ denotes the set of all integers between a and b, including a and b. If $a > b$, we call $[a..b]$ the *empty interval.*

Finally, by a *range* we mean an expression of the form

$$x \in I$$

where x is a variable and I is an interval. We abbreviate $x \in [a..b]$ to $x = a$ if $a = b$ and write $x \in \emptyset$ if $a > b$.

In what follows we discuss various rules that allow us to manipulate linear constraints over interval domains. We assume that all considered linear constraints have at least one variable.

3.1 Reduction Rules for Inequality Constraints

Using appropriate transformation rules each inequality constraint can be equivalently written in the form

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \tag{4}$$

where

- a_i is a positive integer for $i \in POS \cup NEG$,
- x_i and x_j are different variables where $i \neq j$ and $i, j \in POS \cup NEG$,
- b is an integer.

Assume the ranges

$$x_i \in [l_i..h_i]$$

for $i \in POS \cup NEG$.

Choose now some $j \in POS$ and let us rewrite (4) as

$$x_j \leq \frac{b - \sum_{i \in POS-\{j\}} a_i x_i + \sum_{i \in NEG} a_i x_i}{a_j}$$

Computing the maximum of the expression on the right-hand side w.r.t. the ranges of the involved variables we get

$$x_j \leq \alpha_j$$

where

$$\alpha_j := \frac{b - \sum_{i \in POS-\{j\}} a_i l_i + \sum_{i \in NEG} a_i h_i}{a_j}$$

so, since the variables assume integer values,

$$x_j \leq \lfloor \alpha_j \rfloor.$$

We conclude that

$$x_j \in [l_j..min(h_j, \lfloor \alpha_j \rfloor)].$$

By analogous calculations we conclude for $j \in NEG$

$$x_j \geq \lceil \beta_j \rceil$$

where

$$\beta_j := \frac{-b + \sum_{i \in POS} a_i l_i - \sum_{i \in NEG - \{j\}} a_i h_i}{a_j}$$

In this case we conclude that

$$x_j \in [max(l_j, \lceil \beta_j \rceil)..h_j].$$

This brings us to the following reduction rule for inequality constraints:

LINEAR INEQUALITY 1

$$\frac{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in [l_1..h_1], \ldots, x_n \in [l_n..h_n] \rangle}{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in [l'_1..h'_1], \ldots, x_n \in [l'_n..h'_n] \rangle}$$

where for $j \in POS$

$$l'_j := l_j, \ h'_j := min(h_j, \lfloor \alpha_j \rfloor)$$

and for $j \in NEG$

$$l'_j := max(l_j, \lceil \beta_j \rceil), \ h'_j := h_j.$$

3.2 Reduction Rules for Equality Constraints

Each equality constraint can be equivalently written as two inequality constraints. By combining the corresponding reduction rules for these two inequality constraints we obtain a reduction rule for an equality constraint. More specifically, each equality constraint can be equivalently written in the form

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i = b \tag{5}$$

where we adopt the conditions that follow (4) in in the previous subsection.

Assume now the ranges

$$x_1 \in [l_1..h_1], \ldots, x_n \in [l_n..h_n].$$

We infer then both the conclusion of the *LINEAR INEQUALITY 1* reduction rule and

$$x_1 \in [l''_1..h''_1], \ldots, x_n \in [l''_n..h''_n]$$

where for $j \in POS$

$$l''_j := max(l_j, \lceil \gamma_j \rceil), \ h''_j := h_j$$

with

$$\gamma_j := \frac{b - \sum_{i \in POS - \{j\}} a_i h_i + \sum_{i \in NEG} a_i l_i}{a_j}$$

and for $j \in NEG$

$$l''_j := l_j, \ h''_j := min(h_j, \lfloor \delta_j \rfloor)$$

with

$$\delta_j := \frac{-b + \sum_{i \in POS} a_i h_i - \sum_{i \in NEG - \{j\}} a_i l_i}{a_j}$$

This yields the following reduction rule:

LINEAR EQUALITY

$$\frac{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i = b \ ; \ x_1 \in [l_1..h_1], \ldots, x_n \in [l_n..h_n] \rangle}{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i = b \ ; \ x_1 \in [l'_1..h'_1], \ldots, x_n \in [l'_n..h'_n] \rangle}$$

where for $j \in POS$

$$l'_j := max(l_j, \lceil \gamma_j \rceil), \ h'_j := min(h_j, \lfloor \alpha_j \rfloor)$$

and for $j \in NEG$

$$l'_j := max(l_j, \lceil \beta_j \rceil), \ h'_j := min(h_j, \lfloor \delta_j \rfloor).$$

As an example of the use of the above reduction rule consider the CSP $\langle 3x - 5y = 4 \ ; \ x \in [0..9], y \in [1..8] \rangle$. A straightforward calculation shows that $x \in [3..9], \ y \in [1..4]$ are the ranges in the conclusion of *LINEAR EQUALITY* rule. Another application of the rule yields the ranges $x \in [3..8]$ and $y \in [1..4]$ upon which the process stabilizes. In fact, both $x = 3, y = 1$ and $x = 8, y = 4$ are solutions of the equation $3x - 5y = 4$, so the ranges cannot be reduced anymore.

Note that if in (5) there is only one variable, the *LINEAR EQUALITY* rule reduces to the following solving rule:

$$\frac{\langle ax = b \ ; \ x \in [l..h] \rangle}{\langle \ ; \ x \in \{\frac{b}{a}\} \cap [l..h] \rangle}$$

So, if a divides b and $l \leq \frac{b}{a} \leq h$, the domain expression $x = \frac{b}{a}$ is inferred, and otherwise a failure is reached.

3.3 Transformation Rules for Inequality and Equality Constraints

The above reduction rules can be applied only to inequality and equality constraints that are in a specific form, (4) or (5). So we need to augment the introduced reduction rules by appropriate transformation rules. Depending on the level of description one can content oneself with a couple of general rules or several very detailed ones. These rules are pretty straightforward and are omitted.

3.4 Rules for Disequality Constraints

The reduction rules for simple disequalities are very natural. First, note that the following rule

SIMPLE DISEQUALITY 1

$$\frac{\langle x \neq y \ ; \ x \in [a..b], y \in [c..d] \rangle}{\langle \ ; \ \rangle}$$

where $b < c$ or $d < a$, is an instance of the *DISEQUALITY 2* solving rule introduced in Subsection 2.2 and where following the convention there mentioned we dropped the constraint and the domain expressions from the conclusion of the proof rule.

Next, we adopt the following two solving rules that are instances of the solving *DISEQUALITY 3* rule:

SIMPLE DISEQUALITY 2

$$\frac{\langle x \neq y \ ; \ x \in [a..b], y = a \rangle}{\langle \ ; \ x \in [a+1..b] \rangle}$$

SIMPLE DISEQUALITY 3

$$\frac{\langle x \neq y \ ; \ x \in [a..b], y = b \rangle}{\langle \ ; \ x \in [a..b-1] \rangle}$$

and similarly with $x \neq y$ replaced by $y \neq x$. Recall that the domain expression $y = a$ is a shorthand for $y \in [a..a]$.

To deal with disequality constraints that are not simple ones we use the following notation. Given a linear expression s and a sequence of ranges involving all the variables of s we denote by s^- the minimum s can take w.r.t. these ranges and by s^+ the maximum s can take w.r.t. these ranges. The considerations of Subsection 3.1 show how s^- and s^+ can be computed.

We now introduce the following transformation rule for non-simple disequality constraints:

DISEQUALITY 3

$$\frac{\langle s \neq t \ ; \ \mathcal{DE} \rangle}{\langle x \neq t, \ x = s \ ; \ x \in [s^-..s^+] \rangle}$$

where

- s is not a variable,
- x is a fresh variable,
- $\mathcal{DE}$ is a sequence of the ranges involving the variables present in s and t,
- s^- and s^+ are computed w.r.t. the ranges in $\mathcal{DE}$.

An analogous rule is introduced for the inequality $s \neq t$, where t is not a variable.

3.5 Domain splitting rules

We conclude our treatment of linear constraints over interval domains by presenting three specific domain splitting rules. They are so designed that the domains remain intervals:

INTERVAL SPLITTING 1

$$\frac{x \in [a..b]}{x = a \mid x \in [a+1..b]}$$

where $a < b$,

INTERVAL SPLITTING 2

$$\frac{x \in [a..b]}{x = b \mid x \in [a..b-1]}$$

where $a < b$,

INTERVAL SPLITTING 3

$$\frac{\langle x \neq c \ ; \ x \in [a..b] \rangle}{\langle \ ; \ x \in [a..c-1] \rangle \mid \langle \ ; \ x \in [c+1..b] \rangle}$$

where $a < c < b$.

Finally, to deal with the linear constraints the relation symbol of which is $<, \geq$ or $>$ it suffices to provide obvious transformation rules that reduce them to inequalities, disequalities and/or equalities.

3.6 Shifting from Intervals to Finite Domains

In our presentation we took care that all the rules preserved the property that the domains are intervals. In some systems, such as ECL^{i}PS^e, this property is relaxed and instead of finite intervals finite sets of integers are chosen. To model the use of such finite domains it suffices to modify some of the rules introduced above.

In the case of inequality constraints we can use the following minor modification of the *LINEAR INEQUALITY 1* reduction rule:

$$\textit{LINEAR INEQUALITY 2}$$

$$\frac{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in [l'_1..h'_1] \cap D_1, \ldots, x_n \in [l'_n..h'_n] \cap D_n \rangle}$$

where l'_j and h'_j are defined as in the *LINEAR INEQUALITY 1* reduction rule with $l_j := min(D_j)$ and $h_j := max(D_j)$.

Note that in this rule the domains are now arbitrary finite sets of integers. An analogous modification can be introduced for the case of the reduction rule for equality constraints.

In the case of a simple disequality constraint we use the *DISEQUALITY 3* solving rule. So now, in contrast to the case of interval domains, an arbitrary element can be removed from a domain, not only the "boundary" one.

3.7 Example: the *SEND + MORE = MONEY* Puzzle

We now illustrate use of the above rules for linear constraints over interval and finite domains by analyzing in detail the well-known *SEND + MORE = MONEY* puzzle. Recall that this puzzle calls for a solution of the equality constraint

$$\begin{array}{rl} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ + & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = & 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{array}$$

together with 28 simple disequality constraints $x \neq y$ for $x, y \in \{S, E, N, D, M, O, R, Y\}$ where x preceeds y in the alphabetic order, and with the range $[1..9]$ for S and M and the range $[0..9]$ for the other variables.

Both in the CHIP system (see van Hentenryck (1989, page 143)) and in ECL^{i}PS^e Version 3.5.2. the above CSP is internally reduced to the one with the following domain expressions:

$$S = 9, E \in [4..7], N \in [5..8], D \in [2..8], M = 1, O = 0, R \in [2..8], Y \in [2..8]. \tag{6}$$

We now show how this outcome can be formally derived using the rules we introduced.

First, using the transformation rules for linear constraints we can transform the above equality to

$$9000 \cdot M + 900 \cdot O + 90 \cdot N + Y - (91 \cdot E + D + 1000 \cdot S + 10 \cdot R) = 0.$$

Applying the *LINEAR EQUALITY* reduction rule with the initial ranges we obtain the following sequence of new ranges:

$$S = 9, E \in [0..9], N \in [0..9], D \in [0..9], M = 1, O \in [0..1], R \in [0..9], Y \in [0..9].$$

At this stage a subsequent use of the same rule yields no new outcome. However, by virtue of the fact that $M = 1$ we can now apply the *SIMPLE DISEQUALITY 3* solving rule to $M \neq O$ to conclude that $O = 0$. Using now the facts that $M = 1, O = 0, S = 9$, the solving rules *SIMPLE DISEQUALITY 2* and *3* can be repeatedly applied to shrink the ranges of the other variables. This yields the following new sequence of ranges:

$$S = 9, E \in [2..8], N \in [2..8], D \in [2..8], M = 1, O = 0, R \in [2..8], Y \in [2..8].$$

Now five successive iterations of the *LINEAR EQUALITY* reduction rule yield the following sequences of shrinking ranges of E and N with other ranges unchanged:

$$E \in [2..7], N \in [3..8],$$

$$E \in [3..7], N \in [3..8],$$

$$E \in [3..7], N \in [4..8],$$

$$E \in [4..7], N \in [4..8],$$

$$E \in [4..7], N \in [5..8],$$

upon which the reduction process stabilizes. At this stage the solving rules for disequalities are not applicable either.

So using the reduction rules we reduced the original ranges to (6). The derivation, without counting the initial applications of the transformation rules, consists of 24 steps.

Using the *SUBSTITUTION* rule of Subsection 2.2 and obvious transformation rules that deal with bringing the equality constraints to the form (5), the original equality constraint gets reduced to

$$90 \cdot N + Y - (91 \cdot E + D + 10 \cdot R) = 0.$$

Moreover, ten simple disequality constraints between the variables E, N, D, R and Y are still present.

Further progress can now be obtained only by employing a splitting rule. The behaviour of the ECL^{i}PS^e finite domain solver is modelled by the *ENUMERATION* rule of Subsection 2.2. It can be shown, by mimicking the ECL^{i}PS^e execution, that, when the rules here presented are augmented by this rule, there exists a successful derivation for the original CSP representing the *SEND + MORE = MONEY* puzzle.

3.8 Discussion

This concludes our presentation of proof rules that can be used to build a constraint solver for linear constraints over interval and finite domains.

All the rules here introduced are completely straightforward. Still, surprisingly, some of them are not present in ECL^{i}PS^e, Version 3.5.2. In particular, the *SIMPLE DISEQUALITY 1* solving rule is not incorporated and the query `X::0..1, Y::2..3, X ## Y` that corresponds to the CSP $\langle x \neq y \ ; \ x \in [0..1], y \in [2..3]\rangle$ is not solved in ECL^{i}PS^e.

It is worthwhile to mention that the reduction rules *LINEAR INEQUALITY 1* and *LINEAR EQUALITY* are simple modifications of the reduction rule introduced in Davis (1987, page 306) that dealt with closed intervals of reals.

4 Building a Constraint Solver: an Example

We now show how one can use our approach to define specific constraint solvers. By means of example consider a constraint *exactly(x, l, z)* that states that for a list l of variables ranging over some fixed domain exactly x of its elements equal z. We assume that this primitive can be used with x a variable ranging over a subset of natural numbers and with z a variable with an unspecified domain. (This is not a restriction since we can augment the rules below with some natural variable introduction rules.) The *exactly(x,l,z)* primitive is useful for dealing with scheduling problems.

Below we assume that $m > 0$, $i \in [1..m]$ and that $\mathcal{DE}$ stands for a sequence of domain expressions involving the relevant variables. Further, given a set D of natural numbers, we define

$$D - 1 := \{d - 1 \mid d \in D, d > 0\}.$$

The behaviour of the *exactly(x,l,z)* primitive is described by means of the following four rules:

EXACTLY 1

$$\frac{\langle \textit{exactly}(x, [y_1, \ldots, y_m], z) \ ; \ x \in D_x, \mathcal{DE}\rangle}{\langle \textit{exactly}(x, [y_1, \ldots, y_m], z) \ ; \ x \in D_x - \{d \mid d > m\}\rangle}$$

EXACTLY 2

$$\frac{\langle \textit{exactly}(x, [y_1, \ldots, y_m], z) \ ; \ y_i \in D_i, z \in D_z, \mathcal{DE}\rangle}{\langle \textit{exactly}(x, [y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m], z) \ ; \ \rangle}$$

where $D_i \cap D_z = \emptyset$ and $m > 1$,

EXACTLY 3

$$\frac{\langle \textit{exactly}(x, [y_1, \ldots, y_m], z) \ ; \ x \in D_x, y_i = a, z = a, \mathcal{DE}\rangle}{\langle \textit{exactly}(u, [y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m], z), u = x - 1 \ ; \ u \in D_x - 1\rangle}$$

EXACTLY 4

$$\frac{\langle \textit{exactly}(x, [y_1, \ldots, y_m], z) \ ; \ x = 0, \mathcal{DE}\rangle}{\langle y_1 \neq z, \ldots, y_m \neq z \ ; \ \rangle}$$

It is straightforward to see that these four rules are equivalence preserving. These rules, when augmented with a modification of the *LINEAR EQUALITY* rule for finite domains, *DISEQUALITY 1, 2* and *3* rules and the *ENUMERATION* rule, form a stand alone constraint solver.

Such a solver can be directly used for example to solve Latin square puzzles. More importantly, the *exactly(x,l,z)* primitive can in turn be used to specify other constraint primitives that are useful for solving scheduling problems, such as the well-known *atmost(x,l,z)* primitive that states that for a list l of variables ranging over some fixed domain atmost x of its elements equal z. To this end it suffices to adopt the following rule

$$\frac{\langle \textit{atmost}(x, l, z) \ ; \ x \in D_x, \mathcal{DE}\rangle}{\langle \textit{exactly}(y, l, z), y \leq x \ ; \ y \in D_x\rangle}$$

and add an instance of the *LINEAR INEQUALITY 2* rule that deals with simple inequalities of the form $y \leq x$.

5 Conclusions

5.1 Summing up

We presented here a proof theoretic framework that allows us to model computing using constraints.

In general, constraint programming consists of a generation of constraints and of solving them. Both phases can be intertwined. In our presentation we only concentrated on the latter aspect of constraint programming. To complete the picture the framework here presented should be combined with a specific "host" programming language from which the constraints can be generated. To model computing in such an amalgamated language the proof rules should be combined with transitions dealing with the program state. In such a constraint programing language one can distinguish two computation steps.

- If a "conventional" programming statement (such as a procedure call or a built-in in the logic programming framework, or an assignment or a `WHILE` loop in the imperative programming framework) is encountered, a usual transition is performed and the program state is modified accordingly.
- If a constraint is encountered, it is added to the current set of constraints ("constraint store"). This addition is followed by a repeated application of the proof rules to the constraint store. The order of application of these rules is determined by some built-in scheduler (see the next subsection). The terminating condition depends on specific applications.

Further, in such an amalgamated language the interaction between the constraint store and the program state should be properly taken care of.

In the case of linear constraints on finite domains the deterministic proof rules are repeatedly applied until all constraints are solved or a CSP is generated that is closed under the applications of these rules. In the case of algebraic constraints on real intervals the proof rules are repeatedly applied until all constraints are solved or all intervals are smaller than some fixed in advance ϵ.

In some constraint programming languages or in the case of some constraint solvers the splitting rules are not scheduled. Instead, their application is explicitly triggered by some programming construct or facility present in the language.

Let us compare now in more detail our approach to that of Castro (1997) and Frühwirth (1995). In Castro (1997) the proof rules are represented as rewrite rules in the programming language ELAN of Comon, Dincbas, Jouannaud & Kirchner (1995). ELAN allows one to define specific strategies that can be used to schedule these rewrite rules. We defer discussion of this aspect to the next subsection.

In constraint handling rules (CHRs) of Frühwirth (1995) the rules are not meant to maintain equivalence. Further, these rules are not supposed to be used "stand alone" but rather to augment constraint logic programming that provides already a support for the "don't know" nondeterminism. So no splitting rules are available. A minor point is that these rules manipulate constraints only, so the domains need to be encoded as unary constraints.

So the implementation of constraint solvers defined by the ELAN rules and by CHRs is automatically provided by the interpreter, respectively compiler of the language. In contrast, our approach is not geared towards direct implementability even though an implementation specified by such rules is pretty obvious. This allows us to be more abstract and permits us to analyze and construct specific, domain dependent, constraint solvers. As a result we can readily define proof rules that involve some auxiliary computations such as the reduction rules *LINEAR INEQUALITY 1* and *2*. Both ELAN rules and CHRs seem to be too specific to program such proof rules in a direct way.

5.2 Control

One of the issues conspicuously absent in our considerations is that of control. To draw the analogy with the "Algorithm = Logic + Control" slogan of Kowalski (1979), what we defined here is only the "Logic" part of constraint programming. The picture is completed once we can adequately deal with the "Control" part.

In the proof theoretic framework here presented the issue of control enters the picture at three places. First, one needs to schedule the introduced proof rules. Second, one should be able to define which rules are to be scheduled. Finally, one needs some search strategy to traverse the generated proof trees in search for a successful derivation.

The scheduling of the proof rules could be done using a built-in strategy that combines scheduling of the deterministic rules by means of a generic chaotic iteration algorithm of Apt (1997) with the requirement that the applications of the splitting rules are delayed as much as possible. We noted in Apt (1997) that several constraint propagation algorithms employ in fact such a generic algorithm. Further, delay of the applications of the splitting rules prevents unnecessary creation of alternative branches and is a well-known and widely used heuristic. So such a "hard-wired" strategy seems perfectly reasonable.

In contrast, we think that both the selection of specific proof rules and the search strategies for traversing the proof trees should be programmable. What we need here is a programming notation that could allow us to define most common search strategies in a simple way. One possibility would be to use ELAN that, as already mentioned, allows one to define various search strategies. Castro (1997) showed how several of them, such as forward checking and various forms of look ahead, can be implemented in ELAN. We believe that more work is needed to see whether other strategies such as backjumping can be expressed in ELAN, as well.

Another work that should be mentioned here is Schulte (1997) where it is shown how the concept of so-called computation spaces can be used to program in a simple way various search strategies in the programming language Oz (see Smolka (1995)).

Acknowledgements

We would like to thank Eric Monfroy for numerous discussions on the subject of this paper and Carlos Castro and Martin Henz for providing useful comments on a draft of this paper.

References

Andrews, J. H. (1995), 'Foundational issues in implementing constraint logic programming systems', *Science of Computer Programming* **25**(2 & 3), 117–148.

Apt, K. R. (1997), From chaotic iteration to constraint propagation, *in* P. Degano, R. Gorrieri & A. Marchetti-Spaccamela, eds, 'Proceedings of the 24th International Colloquium, ICALP '97', Vol. 1256 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 36–55. Invited Lecture.

Apt, K. R. (1998), 'A logical analysis of Boolean constraints'. Submitted for publication.

Caseau, Y. & Laburthe, F. (1996), Introduction to the CLAIRE programming language, Technical report, Departement Mathématiques et Informatique, Ecole Normale Supérieure, Paris, France.

Castro, C. (1997), 'Constraint manipulation using rewrite rules and strategies'. A Manuscript.

Comon, H., Dincbas, M., Jouannaud, J.-P. & Kirchner, C. (1995), 'A methodological view of constraint solving'. To appear in the Constraints journal.

Davis, E. (1987), 'Constraint propagation with interval labels', *Artificial Intelligence* **32**(3), 281–331.

Frühwirth, T. (1995), Constraint Handling Rules, *in* A. Podelski, ed., 'Constraint Programming: Basics and Trends', LNCS 910, Springer-Verlag, pp. 90–107. (Châtillon-sur-Seine Spring School, France, May 1994).

Hentenryck, P. V., Saraswat, V. & et al. (1996), 'Strategic directions in constraint programming', *ACM Computing Surveys* **28**(4), 701–726.

Jaffar, J. & Maher, M. (1994), 'Constraint logic programming: A survey', *Journal of Logic Programming* **19,20**, 503–581.

Kowalski, R. A. (1979), 'Algorithm = logic + control', *Communications of the ACM* **22**(7), 424–435.

McAloon, K. & Tretkoff, C. (1995), 2LP: Linear programming and logic programming, *in* P. V. Hentenryck & V. Saraswat, eds, 'Principles and Practice of Constraint Programming', MIT Press, pp. 101–116.

Schulte, C. (1997), Programming constraint inference engines, *in* G. Smolka, ed., 'Proceedings of the 3rd International Conference on Constraint Programming (CP97)', Lecture Notes in Computer Science, vol. 1330, Springer-Verlag, Berlin, pp. 519–533.

Sieber, S. M., Schabes, Y. & Pereira, F. C. N. (1995), 'Principles and implementation of deductive parsing', *Journal of Logic Programming* **24**(1 & 2), 3–36.

Smolka, G. (1995), The Oz programming model, *in* J. van Leeuwen, ed., 'Computer Science Today', Lecture Notes in Computer Science, vol. 1000, Springer-Verlag, Berlin, pp. 324–343.

Smolka, G. (1996), 'Problem solving with constraints and programming', *ACM Computing Surveys* **28**(4es), . Electronic Section.

van Emden, M. H. (1997), 'Value constraints in the CLP scheme', *Constraints* **2**(2), 163–184.

van Hentenryck, P. (1989), *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, Cambridge, MA.

Wetzel, G., Kowalski, R. & Toni, F. (1995), A theorem-proving approach to CLP, *in* A. Krall & U. Geske, eds, '11th Workshop on Logic Programming', GMD-Studien Nr. 270, TU Wien.