Extending clp(FD) by Negative Constraint Solving

Antonio J. Fernández and Patricia M. Hill

Abstract

In this paper we present a novel extension to the constraint logic programming on Finite Domains (clp(FD)) which combines the usual clp(FD) primitive constraint $X\ in\ range$ with a new primitive constraint $X\ notin\ range$. We show that, together, they are more expressive and provide better constraint propagation than having $X\ in\ range$ on its own.

It is shown how the standard clp(FD) framework can be adapted to support this extension by defining conditions for the checking of the consistency of constraints and new methods for their propagation.

Keywords: Constraint Solving, Constraint Propagation, Labeling, Finite Domain,

1 Introduction

The finite domain constraint solver clp(FD) has a simple framework based on a unique primitive of the form $X\ in\ r$ where X is a variable and r a subset of the integer domain. In [1], it is shown that this framework allows a highly efficient constraint solver with good expressivity.

The clp(FD) framework is closed under negation. This means that that the constraint $\neg(X\ in\ r)$ is just the complement $X\ in\ -r$ where $-r = \{i \in Integer \mid i \notin r\}$. This negative constraint highlights two problems: from a computational point of view, a negative constraint is expensive since it will introduce choice points in the solution search tree and, from a programming point of view, it does not provide much flexibility since it must be defined in terms of the range r and not the complement. For example, to say that X does not belong to the subset $\{i \in Natural \mid 3 \leq i \leq 5\}$ we must write $X\ in\ \{i \in Natural \mid (0 \leq i\ < 3) \text{ or } (5 < i \leq \infty\}$.

Typically the constraint satisfaction problems on finite domains (FD) involve several processes: constraint propagation, consistency checks, and labeling (also called enumeration). Constraint propagation entails updating the constraint store with new constraints. The revised constraint store may be checked for consistency.

Antonio J. Fernández is with Departamento de Lenguajes y Ciencias de la E.T.S.I.I., 29071 Teatinos, Málaga. Email: `afdez@lcc.uma.es`

Patricia M. Hill is with School of Computer Studies,University of Leeds, England. Email: `hill@scs.leeds.ac.uk`

This work was partly supported by EPSRC grants GR/L19515 and GR/M05645 and by CICYT grant TIC98-0445-C03-03.

Labeling assigns domain values to some of the finite domain variables in order to solve the constraints.

In this paper, we propose an extension to the clp(FD) framework based on indexicals. This extension provides a new primitive constraint X *notin* r which captures the meaning of the negated constraint $\neg(X\ in\ r)$ in the standard $clp(FD)$ framework. However, we also provide a means for these new constraints to actively interact with other constraints of the form X *in* r and, as a consequence, allowing for improved constraint propagation and thereby reducing the labeling needed for finding a solution (by reducing the solution search tree).

Since, as shown here, high level constraints can be defined directly in terms of this primitive, by supporting this new primitive constraint, the expressivity of the clp(FD) framework is also improved.

The paper is structured as follows. Section 2 describes the usual clp(FD) framework based on the constraint X *in* r . In Section 3 we describe the new primitive X *notin* r and explain how it affects to the standard clp(FD). Section 4 shows how constraint propagation is improved and in Section 5 some examples of solvers, which demonstrate the flexibility of the combination of the constraints X *in* r and X *notin* r , are shown. In section 6 a comparison of the current clp(FD) framework with our extension in the solving of several well known problems is done. The paper ends with some conclusions and further work.

2 Basic concepts

2.1 Syntax of X *in* r

The following definitions are from [1].

A domain in FD is a (non empty) set of natural numbers (i.e. a range). A range is a subset of $\{0, 1, \ldots, \mathit{infinite}\}$ where *infinite* is a particular integer denoting the greatest value that a variable can take. A range is consistent if it is not the empty set and inconsistent otherwise. The notation $a..b$ represents the range $\{i \in Natural | a \leq i \leq b\}$. Given a range r, $min(r)$ (resp. $max(r)$) is defined to be the lower (resp. upper) bound of r. *Pointwise operations* (+,-,*,/) between a range r and an integer i are defined as the set obtained by applying the corresponding operation on each element of r. The set of FD variables to have values in a given domain is denoted by V_0.

If X is a variable, then a constraint c is defined as follows:

$$
\begin{array}{rcl}
c ::= & X \ in \ r & (\mathit{primitive\ constraint}) \\
r ::= & t_1..t_2 & (\mathit{interval}) \\
 & -r & (\mathit{complementation}) \\
 & r_1 \backslash/ r_2 & (\mathit{union}) \\
 & r_1 /\backslash r_2 & (\mathit{intersection}) \\
 & r \bullet i & (\mathit{pointwise\ operations};\ i \in \mathit{Integer};\ \bullet \in \{+, *, -, /\} \\
t ::= & \mathit{min}(Y) & (\mathit{indexical\ term\ min}) \\
 & \mathit{max}(Y) & (\mathit{indexical\ term\ max}) \\
 & \mathit{val}(Y) & (\mathit{indexical\ term\ val}) \\
 & ct & (\mathit{constant\ term}) \\
 & t_1 \bullet t_2 & (\mathit{integer\ operations};\ \bullet \in \{+, *, -, /\} \\
ct ::= & \mathit{infinite} & (\mathit{greatest\ value\ in\ the\ domain}) \\
 & n & (n \in 0..\mathit{infinity}) \\
 & ct_1 \bullet ct_2 & (\mathit{integer\ operations};\ \bullet \in \{+, *, -, /\}
\end{array}
$$

This syntax can vary depending on the different languages that employ this approach (usually called indexical approach). Here we have presented only the standard syntax. Intuitively, a constraint $X \ in \ r$ enforces to X to belong to the range r that can be not only a constant range but also an indexical range using:

- $min(X)$ representing the minimum value of the current domain X
- $max(X)$ representing the maximum value of the current domain X
- $val(X)$ representing the delayed value of the current domain X (this value is only returned when X is instantiated).

2.2 Constraint consistency, interval narrowing and constraint propagation

We say that a constraint $X \ in \ r$ is simple if it does not use indexicals terms and non-simple otherwise. For example, $X \ in \ 1..3$ is simple and $X \ in \ min(Y)..max(Y)$ is non-simple.

Definition 1 *(Consistency of constraints) Let $X \ in \ r$ be a simple constraint. We say that $X \ in \ r$ is inconsistent iff r is the empty set and consistent otherwise.*

Definition 2 *(Store) A store is a finite set of constraints. A store is in* normal form *if it contains at most one simple constraint $X \ in \ r$ for each variable belonging to the original set of constrained variables. The normal form of a store S can be obtained by replacing, for each variable $X \in V_0$, the set of all simple constraints for X, $X \ in \ r_1, \ldots, X \ in \ r_n$, by the constraint $X \ in \ r_1 /\backslash \ldots /\backslash r_n$. This process is called* interval narrowing.

We say that a store is inconsistent if it contains at least one inconsistent constraint.

For example, consider the store $S = \{X\ in\ 2..10, Y\ in\ 4..20\}$. By adding the constraint $X\ in\ 5..15$, an interval narrowing process is executed and the new store is $\{X\ in\ 5..10, Y\ in\ 4..20\}$.

When a constraint $X\ in\ r$ uses an indexical term on another variable Y (the indexed variable), it must be checked each time the domain of Y is updated. This process, called *constraint propagation*, projects the value returned by the indexical term defined over Y on the constraint $X\ in\ r$ and a new simple constraint on X is sent to the store. This may cause further interval narrowing on X. The constraints using indexicals remain in the store to propagate future modifications on the indexed variables.

For example, consider the store $S = \{X\ in\ 2..10, Y\ in\ 4..20, Y\ in\ min(Z)..20\}$. By adding the constraint $X\ in\ min(Y)..max(Y)$, constraint propagation enforces the constraint $X\ in\ 4..20$. This constraint is sent again to the store S and, by interval narrowing, the new store is $\{X\ in\ 4..10,\ \ Y\ in\ 4..20,\ \ Y\ in\ min(Z)..20,\ X\ in\ min(Y)..max(Y)\}$.

During computation on a store, a constraint added to it can:

- *succeed*: if the store becomes consistent.
- *fail*: if the store becomes inconsistent.
- *propagate*: if the constraint is non-simple and leads to constraint propagation. The resultant constraint is added to the store.
- *suspend*: if the constraint is non-simple and any of its indexical terms cannot be evaluated so that constraint propagation is delayed until all its indexical terms can be evaluated. For instance, in the example shown above constraint propagation on the constraint $Y\ in\ min(Z)..20$ is delayed until $min(Z)$ is evaluated.

See [1] for more details.

2.3 Negation

The primitive $X\ in\ r$ embeds the core propagation mechanism. High level constraints are built from this primitive. For example, the constraint $X \neq Y/2$ is defined into the clp(fd) system [2] as follows:

$$\begin{aligned} X \neq Y :- \quad & X\ in\ -\{val(Y)\}, \\ & Y\ in\ -\{val(X)\}. \end{aligned}$$

where the constraint $X\ in\ -\{val(Y)\}$ (resp. $Y\ in\ -\{val(X)\}$) is delayed until Y (resp. X) is bound.

The FD constraint system based on indexicals is closed under negation since the constraint $\neg X\ in\ r$ is just the complementation $X\ in\ -r$. Intuitively, this means that a constraint $\neg X\ in\ a..b$, where $a, b \in 0..infinity$, is equivalent to the constraint $X\ in\ 0..a-1 \backslash/ b+1..infinity$. As consequence of it, a constraint $X\ in\ -\{val(Y)\}$

is equivalent to $X\ in\ 0..\{val(Y)\}-1\backslash/\{val(Y)\}+1..infinity$. Note that this kind of constraint introduces a choice point (in form of union of ranges) to the solution search tree.

2.4 Propagation plus labeling

Most of the problems cannot be solved by just constraint propagation and interval narrowing but they need an additional process called labeling (or enumeration). Labeling is the process of assigning to the FD variables, values from the domain by following a given criteria. For example, a naive labeling chooses values from the minimum to the maximum in the domain of a FD variable and assigns these values to the FD variables sequentially. Labeling a FD variable X with a value n introduces in the solution search tree a choice point and two alternatives: $X = n$ and $X \neq n$. Labeling is thus an expensive process which is executed when no more constraint propagation, over the FD variables constrained in the problem, is possible.

Note that any improvement on the constraint propagation leads to a reduction in the solution search tree since less quantity of labeling (measured in the number of assignments to the FD variables that it is necessary to do in order to find a solution or an inconsistency) will be required.

3 An alternative to the complementation of ranges

We propose an extension to the clp(FD) framework which consists in an alternative approach for the negation of the primitive constraint $X\ in\ r$. In next sections, this approach is described as well as its advantages and differences wrt the constraint $X\ in\ -r$.

3.1 The primitive $X\ notin\ r$

We extend the clp(FD) framework by defining a new primitive constraint $X\ notin\ r$. The syntax shown in section 2.1 is extended as follows:

$$
\begin{array}{rcl}
c ::= & X\ in\ r & (\textit{primitive positive constraint}) \\
 & X\ notin\ r & (\textit{primitive negative constraint})
\end{array}
$$

The rest of the syntax remains unchanged. Intuitively, the constraint $X\ notin\ r$ constrains X to have values that do not belong to r. Before explaining how this new primitive affects the clp(FD) framework, we introduce some concepts.

The constraint $X\ in\ r$ is called a *positive constraint* and is denoted as constraint+.

The constraint $X\ notin\ r$ is called a *negative constraint* and is denoted as constraint−.

Simple (resp. non-simple) and consistent (resp. inconsistent) negative interval constraints are defined analogously as for positive constraints.

We denote by $c^s_{\neg X}$ (resp. $c^{ns}_{\neg X}$) a simple (resp. non-simple) negative constraint, defined on a variable X, when necessary. Analogously, we denote by c^s_X (resp. c^{ns}_X) a simple (resp. non-simple) positive constraint, defined on a variable X.

A store containing only negative (resp. positive) constraints is called a negative (resp. positive) constraint store. We write $\neg S$ (resp. $+S$) to express S is a negative (resp. positive) store. The set of all variables constrained at least with a constraint in $\neg S$ (resp. $+S$) is denoted by $V_{\neg S}$ (resp. V_{+S}).

3.2 Considerations on the negative stores

3.2.1 Constraint propagation

Constraint propagation is executed analogously as for positive stores. This means that the constraints$-$ using indexicals terms on other variables remain in the negative store for propagating future changes on the indexed variables.

Consider for example the following stores:

$$+S = \{X\ in\ 10..20\} \text{ and } \neg S = \{X\ notin\ 40..60,\ X\ notin\ 1..max(Y) * 3\}$$

By adding the constraint $Y\ in\ 1..4$ to the store $+S$, the constraint $X\ notin\ 1..max(Y)*3$ is propagated to the constraint $X\ notin\ 1..12$.

3.2.2 Constraint Normalisation

The interval narrowing process is never executed on negative stores and is replaced by a process called constraint normalisation. This process is consequence of the interaction of both a positive and a negative store.

Definition 3 *(Normalised store) Let $+S$ be a positive store in normal form and let $\neg S$ be a negative store. And let $V = \{X \mid X \in (V_{+S} \cap V_{\neg S}) \wedge \exists c^s_{\neg X} \in \neg S\}$ be the set of all variables constrained with precisely one simple constraint+ in $+S$ and at least one simple constraint$-$ in $\neg S$. The* normalised store *of $\neg S$ wrt $+S$ is a negative store defined as follows:*

$$norm(\neg S, +S) = \neg S - \bigcup_{X \in V} \{c^s_{\neg X} \mid c^s_{\neg X} \in \neg S\} \cup \bigcup_{X \in V, X\ notin\ r_2 \in \neg S} \{X\ notin\ r_1 /\backslash r_2) \mid X\ in\ r_1 \in +S \wedge \{r_1 /\backslash r_2\} \neq \emptyset\}$$

This means that $norm(\neg S, +S)$ is the negative store obtained by

(1) removing in $\neg S$ all the simple constraints$-$ defined on variables which are constrained by a simple constraint+ in $+S$, and of

(2) adding certain consistent simple constraints$-$ which are the result of

the intersection of the ranges of the simple constraint+ in $+S$ for a variable X and a simple constraint$-$ in $\neg S$ for the same variable X.

Consider for example the following stores:

$$+S = \{X\ in\ 10..20\} \text{ and } \neg S = \{X\ notin\ 40..60,\ X\ notin\ 15..30\}$$

Then $norm(\neg S, +S) = \{X\ notin\ 15..20\}$. The constraints $X\ notin\ 40..60$ and $X\ notin\ 15..30$ were removed from $\neg S$ whereas the constraint $X\ notin\ 15..20$ (which is the result of $X\ notin\ (10..20) /\backslash (15..30)$) was added.

3.3 Consistency checking wrt negative constraints

The consistency of a positive store can be directly checked wrt a negative store.

Definition 4 *(Consistency of a variable wrt a negative store) Let $+S$ be a positive store in normal form and let X in r_1 the simple interval constraint in $+S$ for any variable $X \in V_{+S}$. Let $\neg S$ be a negative store. Then X is* inconsistent *in $+S$ wrt $\neg S$ if*

$$\exists (X \ notin \ r_2) \in norm(\neg S, +S) : \ r_1 \subseteq r_2$$

and *consistent* otherwise.

For example, consider the following stores:

$$+S = \{X \ in \ 3..6, \ Y \ in \ 10..20\} \text{ and } \neg S = \{X \ notin \ 5..10, \ Y \ notin \ 8..22\}$$

Then X is consistent in $+S$ wrt $\neg S$ and Y is inconsistent in $+S$ wrt $\neg S$.

Definition 5 *(Consistency of a positive store wrt a negative store) Let $+S$ and $\neg S$ be a positive and a negative store respectively where $+S$ is in normal form. Then $+S$ is consistent wrt $\neg S$ iff for all $X \in V_{+S}$, X is consistent in $+S$ wrt $\neg S$, and inconsistent otherwise.*

4 Improving constraint propagation

In this section we show how the interaction of negative and positive constraints leads to the improvement of the constraint propagation.

4.1 The disjoint element

Definition 6 *(The disjoint range) Let $r_1 = t_1..t_2$ and $r_2 = t_3..t_4$ be two consistent simple ranges. We call the* disjoint *range of r_1 and r_2 and denoted by $\mathcal{D}_{r_1,r_2}$, to the range defined as follows:*

$$\begin{array}{rl}
if \ (r_1 /\backslash r_2 = \emptyset) \ then & \mathcal{D}_{r_1,r_2} = \ r_1 \\
elsif \ (t_1 \leq t_3) \ and \ (t_2 < t_4) & \mathcal{D}_{r_1,r_2} = \ t_1..t_3 - 1 \\
elsif \ (t_3 \leq t_1) \ and \ (t_4 < t_2) & \mathcal{D}_{r_1,r_2} = \ t_4 + 1..t_2 \\
otherwise & it \ does \ not \ exist
\end{array}$$

Examples. Table 1 shows some examples of the disjoint range for two ranges.

Table 1: Disjoint ranges. Examples

r_1	r_2	$\mathcal{D}_{r_1,r_2}$
1..15	10..20	1..9
10..20	1..15	16..20
4..7	9..11	4..7
1..10	5..7	it does not exist
1..10	0..20	it does not exist

Definition 7 *(The Disjoint constraint) Let $c_1 = X \ in \ r_1$ and $c_2 = X \ notin \ r_2$ be respectively a positive and a negative simple constraint defined on the same variable X. The* disjoint constraint *of c_1 and c_2 is defined as the constraint X in $\mathcal{D}_{r_1,r_2}$ and is denoted as $\mathcal{D}_{c_1,c_2}$.*

4.2 Further propagation by extending the positive store

On FDs, when no more constraint propagation is possible an enumeration (labeling) process is activated. As explained in section 2.4, the assignment of domain values to the FD variables introduces backtracking in the form of choice points with the corresponding penalty in efficiency.

The importance of the disjoint element is that it can be used to get more propagation before labeling is activated. In the following, we define the concept of *extended store* and show, by means of a simple example, how more constraint propagation can be got by extending the positive store.

Definition 8 *(Extended store) Let $+S$ and $\neg S$ be a positive and a negative store respectively where $+S$ is in normal form. Consider the positive store S' defined as follows*

$$\begin{array}{rl} S' = & \{c_X^{ns} \mid c_X^{ns} \in +S\} \\ \cup & \{c_X^{s} \mid c_X^{s} \in +S \wedge \neg\exists c_{\neg X}^{s} \in norm(\neg S, +S)\} \\ \bigcup_{c_{\neg X}^{s} \in \neg S} & \{\mathcal{D}_{c_X^s, c_{\neg X}^s} \mid c_X^s \in +S \ \wedge \mathcal{D}_{c_X^s, c_{\neg X}^s} \ exists\} \\ \bigcup_{c_{\neg X}^{s} \in \neg S} & \{c_X^s \mid c_X^s \in +S \ \wedge \mathcal{D}_{c_X^s, c_{\neg X}^s} \ does \ not \ exist\} \end{array}$$

The normal form of S' is called the extended store *of $+S$ wrt $\neg S$ and is denoted by $ext(+S, \neg S)$.*

For instance, consider the following stores:

$$+S = \{X \ in \ 10..20, Y \ in \ 1..3\} \text{ and } \neg S = \{X \ notin \ 3..11, \ X \ notin \ 15..30\}$$

As

1. $norm(\neg S, +S) = \{X \ notin \ 10..11, X \ notin \ 15..20\}$;
2. $\mathcal{D}_{X \ in \ 10..20, X \ notin \ 3..11} = \ X \ in \ 12..20$;
3. $\mathcal{D}_{X \ in \ 10..20, X \ notin \ 15..30} = \ X \ in \ 10..14$;
4. $S' = \{X \ in \ 12..20, X \ in \ 10..14, Y \ in \ 1.., 3\}$.

By the definition shown above, $ext(+S, \neg S)$ is the normal form of S', that is, $\{X \ in \ 12..14, Y \ in \ 1..3\}$.

Note that the constraint $X \ in \ 10..20$ belonging to $+S$ has been narrowed, by extending the positive store, to the constraint $X \ in \ 12..14$. Note also that no additional choice point (backtracking) has been added in order to get such a reduction in the range.

4.3 Negative Constraints vs. Complemented Constraints

Although in their meanings, a constraint $X \ notin \ r$ is equivalent to the constraint $X \ in \ -r$, these constraints are solved in different ways. For instance, consider the example shown above where each negative constraint in $\neg S$ is now replaced in $+S$ by their equivalent complemented constraint. $+S$ becomes:

$$+S = \{X \ in \ -(3..11), \ X \ in \ -(15..30), \ Y \ in \ 1..3\}$$

This means that (see Section 2.3)

$$+S = \{X\ in\ 10..20,\ X\ in\ (0..2)\backslash/(12..infinity),\ X\ in\ (0..14)\backslash/(31..infinity),\ Y\ in\ 1..3\}$$

By interval narrowing, $+S$ is expressed:

$$\begin{array}{lll} +S = \{ & X\ in & ((10..20)/\backslash(0..2)/\backslash(0..14))\ \backslash/ \\ & & ((10..20)/\backslash(0..2)/\backslash(31..infinity))\ \backslash/ \\ & & ((10..20)/\backslash(12..infinity)/\backslash(0..14))\ \backslash/ \\ & & ((10..20)/\backslash(12..infinity)/\backslash(31..infinity)), \\ & Y\ in & 1..3\} \end{array}$$

The resulting store is $+S = \{X\ in\ 12..14,\ Y\ in\ 1..3\}$ which is exactly the same solution got by extending the positive store in previous section.

However, note that the use of negative constraints did not introduce choice points whereas the use of complemented constraints introduces additional choice points (in the way of union of ranges) to the solution search tree. This example shows how the use of negative constraints instead of complemented constraints can avoid unnecessary choice points and thus reduce the cost of finding a solution or an inconsistency.

5 Defining constraints by means of negative ones

In usual clp(FD), high level constraints (i.e the constraint $X \neq Y$) are built from the constraint $X\ in\ r$ at a user level so that the user controls the propagation mechanism. One of the main advantages of the interaction of the constraint $X\ in\ r$ and $X\ notin\ r$ is that the user can also control the propagation with *notin* constraints.

In the following we present some examples which demonstrate the flexibility and expressivity of the integration of the primitive $X\ notin\ r$ in the usual clp(FD).

5.1 Symbolic constraints: The all_different constraint

In most clp(FD) systems, many symbolic constraints are hard-coded in C to obtain the best performance so that the user has no control over the propagation mechanism.

As an example, we consider the *all_different/1* constraint which is applied to a list L of FD variables and enforces the constraint $X \neq Y$ for each pair X, Y of variables in the list L.

This symbolic constraint has a number of alternative implementations. For example, in the clp(FD) system [2], it depends exclusively on the high level constraint $X \neq Y$ which is defined there as:

$$\begin{array}{lll} X \neq Y & :- & X\ in\ -\{val(Y)\}, \\ & & Y\ in\ -\{val(X)\}. \end{array}$$

The constraint $X\ in\ -\{val(Y)\}$ (resp. $Y\ in\ -\{val(X)\}$) is delayed until Y (resp. X) is bound. This constraint is based on the complementation of ranges which, in the clp(FD) system, is coded in C. We refer to the source code of clp(FD) which is provided with the implementation[1]. Note also that by the complementation of

[1] File fd_range.c.

ranges, a constraint such as $X\ in\ -\{val(Y)\}$ introduces one choice point in the way of $X\ in\ 0..\{val(Y)\}-1 \backslash/ X\ in\ \{val(Y)\}+1..infinity$.

As the constraint *all_different/1* is based on the complementation of ranges, a constraint *all_different(L)* results in a solution search tree which grows with the number of constraints $X \neq Y$ imposed for each $X, Y \in L$.

In other systems, such as SICStus Prolog 3#7.1 [3], the implementation of the *all_different/1* constraint is really hard to follow since it is directly coded in C and very specialised for FDs. The aim is to get an extra efficiency. We remit to the source code of SICStus which is provided with the implementation [2].

We propose a definition based exclusively in the constraint $X \neq Y$, similar to that shown above for the clp(FD) system. The main difference is that this constraint is now defined by the constraint $X\ notin\ r$ instead of the constraint $X\ in\ r$.

$$\begin{aligned} X \neq Y \quad :- \quad & X\ notin\ val(Y), \\ & Y\ notin\ val(X). \end{aligned}$$

5.1.1 Compared definitions

The clp(FD) definition and our definition are straightforward and easier to read than SICStus one which is coded directly in C.

Although our definition is similar to the clp(FD) definition, several differences are highlighted. First, as already mentioned, our definition uses directly the range being negated and not its complement as in the clp(FD) definition. Secondly, as consequence of the complementation of ranges, the clp(FD) definition introduces extra choice points (see Section 2.3) whereas our definition avoids them by the intrinsic meaning of the *notin* constraint.

5.2 Boolean Solvers

Boolean solvers have been used from a long time in various research areas. The idea of considering Boolean solvers as a particular case of FD was first introduced in the language CHIP [5]. However, in CHIP the Boolean solvers were coded by using low level routines. In clp(FD) [1] Boolean solvers were coded directly at the user level by means of the constraint $X\ in\ r$. These Boolean solvers are obviously more readable than the Boolean solvers of CHIP since the schema of propagation is coded in a constraint language and not in C.

In this section we show an alternative definition for the Boolean solvers of the clp(FD) system. These definitions are based on the combined use of the constraints $X\ in\ r$ and $X\ notin\ r$ and highlight the expressive power of our extension. These examples show how, in several cases, the formulation is even more natural than that shown in the clp(FD) system and how the code of these constraints is even more readable.

[2] File fd.c.

5.2.1 Not/1

The clp(FD) system codes the Boolean constraint $not(X, Y)$, which means $X \equiv \neg Y$ as follows:

$$not(X, Y) :- X\ in\ \{1 - val(Y)\},\ Y\ in\ \{1 - val(X)\}.$$

Our alternative definition is as follows:

$$not(X, Y) :- X\ notin\ val(Y),\ Y\ notin\ val(X).$$

Note that our definition is more intuitive since it is expressed directly with the negated range.

5.2.2 Or/3

The Boolean constraint $or(X, Y, Z)$, which means $Z \equiv X \vee Y$, is coded in the clp(FD) system as follows:

$$\begin{aligned} or(X, Y, Z) \quad :- \quad & X\ in\ 0..1, Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & Z\ in\ min(X) + min(Y) - min(X) * min(Y)..max(X) + max(Y) - max(X) * max(Y), \\ & X\ in\ min(Z) * (1 - max(Y))..max(Z), \\ & Y\ in\ min(Z) * (1 - max(X))..max(Z). \end{aligned}$$

Our alternative definition is the following:

$$\begin{aligned} or(X, Y, Z) \quad :- \quad & X\ in\ 0..1,\ Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & X\ in\ 0..val(Z),\ Y\ in\ 0..val(Z), \\ & Z\ notin\ \{val(X) * val(Y)\}, \\ & Z\ notin\ \{1 - val(X) + val(Y)\}, \\ & X\ notin\ \{1 - val(Z) + val(Y)\}, \\ & Y\ notin\ \{1 - val(X) + val(Z)\}. \end{aligned}$$

Note that it is based on the combined use of the constraints $X\ \ in\ r$ and $X\ \ notin\ r$.

5.2.3 And/3

The clp(FD) system codes the Boolean constraint $and(X, Y, Z)$ which means $Z \equiv X \wedge Y$ as:

$$\begin{aligned} and(X, Y, Z) \quad :- \quad & X\ in\ 0..1,\ Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & Z\ in\ min(X) * min(Y)..max(X) * max(Y), \\ & X\ in\ min(Z)..max(Z) * max(Y) + 1 - min(Y), \\ & Y\ in\ min(Z)..max(Z) * max(X) + 1 - min(X). \end{aligned}$$

Our alternative definition is shown below:

$$\begin{aligned} and(X, Y, Z) \quad :- \quad & X\ in\ 0..1,\ Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & Z\ in\ min(X) * min(Y)..max(X) * max(Y), \\ & Y\ in\ min(Z)..1,\ X\ in\ min(Z)..1, \\ & Y\ notin\ val(Z) + 1..val(X),\ X\ notin\ val(Z) + 1..val(Y). \end{aligned}$$

Again note the combined use of the primitive constraints.

5.2.4 Other boolean constraints

The boolean constraint $equiv(X,Y,Z)$, which is true if $Z \equiv X \Leftrightarrow Y$, is coded in the clp(FD) system as follows:

$$\begin{aligned} equiv(X,Y,Z) \quad :- \quad & X\ in\ 0..1,\ Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & Z\ in\ \{1-((val(X)+val(Y))\ mod\ 2)\}, \\ & X\ in\ \{1-((val(Y)+val(Z))\ mod\ 2)\}, \\ & Y\ in\ \{1-((val(X)+val(Z))mod2)\}. \end{aligned}$$

We code it as follows:

$$\begin{aligned} equiv(X,Y,Z) \quad :- \quad & X\ in\ 0..1,\ Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & Z\ notin\ val(X)+val(Y),\ Z\ notin\ val(X)-val(Y), \\ & X\ notin\ val(Z)+val(Y),\ X\ notin\ val(Z)-val(Y), \\ & Y\ notin\ val(Z)+val(X),\ Y\ notin\ val(Z)-val(X). \end{aligned}$$

Other Boolean constraints are easily defined. For example, the Boolean constraint $xor(X,Y,Z)$ is directly defined as follows:

$$\begin{aligned} xor(X,Y,Z) \quad :- \quad & X\ in\ 0..1,\ Y\ in\ 0..1,\ Z\ in\ 0..1, \\ & equiv(X,Y,W), Z\ notin\ val(W), \\ & equiv(X,Z,W1), Y\ notin\ val(W1), \\ & equiv(Z,Y,W2), X\ notin\ val(W2). \end{aligned}$$

The examples defined above show that the combined use of the constraints $X\ in\ r$ and $X\ notin\ r$ provides a way in which the expressivity of the clp(FD) framework increases since the propagation of both positive and negative constraints can be control at the user level.

6 A comparison on the constraint propagation

We have implemented a prototype to check the validity of the negative constraints as described here. This prototype is written in *SICStus* 3#7.1 by using the library *chr* which is based on constraint handling rules [4] and provided with the implementation of *SICStus* 3#7.1. Our prototype is available in the following address *http://www.lcc.uma.es/~afdez/negativeCons.*

The different versions of the *all_different* constraint as well as the Boolean solvers shown in Section 5 were tested on a set of benchmarks. The comparison was focused on the quantity of constraint propagation. All the benchmarks were solved by using the minimum labeling possible: that is to say, by labeling the minimum number of FD variables necessary to solve the problem. This provided a measure of the quantity of constraint propagation. In each the case, we use a naive labeling strategy in which the values were chosen from the minimum to the maximum in the domain. The chosen problems were the following:

- A **Colouring** problem which involves 6 FD variables ranging in the domain 1..3 with 9 constraints in the way of $X \neq Y$.

- The usual **N queens** problem which involves inequality constraints among the FD variables.

- The **Suudoku problem** which requires a 9×9 square to be partially filled in such a way such that each row and column are permutations of $[1, \ldots, 9]$, and each 3×3 square, where the leftmost column modulo 3 is 0, is a permutation of $[1, \ldots, 9]$. This problem makes repetitive use of the *all_different* constraint.

- A **Pigeons-Holes** problem. Put N pigeons in M pigeon-holes. This problem is coded by using the constraints *at_most_one* and *only_one*. These constraint were defined from the Boolean operators *and* and *or*.

- A **circuit diagnosis** problem which uses the Boolean solvers *or*, *and*, *xor* and *equiv* and the inequality constraint $X \neq Y$.

- The **Schur's Lemma** problem. Color the integers $1, 2, \ldots, N$ with 3 colors so that there is no monochrome triplets (x, y, z) where $x + y = z$. This problem uses the Boolean solver *and*/3 as well as the high level constraints *only_one*/1 and *at_most_one*/1 (which were defined by using the Boolean operators *and* and *or*).

Our prototype solved all the benchmarks by using the same minimum quantity of labeling that SICStus and clp(FD) systems. This is surprising since we got the same solutions without using any specific built-in constraints (as the *all_different* constraint) or by use of C code.

We also implemented a prototype which simulated the behaviour of the usual clp(FD). The results showed that the solution search tree was much reduced by using the constraint X *notin* r instead of the constraint X *in* $-r$. This reduction in the number of choice nodes in the solution search tree depends on the number of FD variables involved in the range r. This result was expected since, as already explained, the complementation of ranges introduces in the solution search tree additional choice points.

Unfortunately we could not compare running times since, to do this fairly, we need to implement the primitive X *notin* r at lower level. We plan to do this in future work.

7 Conclusions

In this paper we have described how the clp(FD) framework based on the primitive X *in* r can be extended with a new primitive X *notin* r which captures the meaning of the constraint $\neg X$ *in* r. The resulting clp(FD) framework provides better constraint propagation as well as a way in which the expressivity of the usual clp(FD) framework is increased.

We have shown how the use of the primitive $X\ notin\ r$ reduces the solution search tree with respect to the use of the constraint $\neg X\ in\ r$ by avoiding the addition of the extra choice points which are the result of the disjunction of the ranges introduced by the constraint $X\ in\ -r$.

The conditions for the consistency of the new primitive have been defined in this paper and we have also described how the constraint propagation of the existing clp(FD) framework is affected with the inclusion of this new primitive.

The use the primitive $X\ notin\ r$ and its combination with the primitive $X\ in\ r$ has been illustrated by means of examples. By implementing a prototype, the results shown that constraint propagation improves wrt the usual clp(FD) and that an important reduction in the solution search tree is got. However, we could not measure running times so that we plan to extend the current implementation of the clp(FD) system [2] to include the primitive $X\ notin\ r$. Then we will be able to compare performances.

All the ideas shown in this paper have been tested on a prototype built by means of the library *chr* [4] which is included in the distribution of SICStus Prolog 3#7.1. This prototype is available in the following address $http://www.lcc.uma.es/\sim afdez/negativeCons$.

References

[1] Codognet P. and Diaz D., Compiling Constraints in *clp(FD)*. In *The Journal of Logic Programming*, 27, pp:185-226,1996.

[2] Diaz D., clp(FD) 2.21 User's Manual. 1994.

[3] Carlsson M., Ottosson G. and Carlson B., An Open-Ended Finite Domain Constraint Solver. In Proc. of the *9th International Symposium on Programming Languages: Implementation, Logics and Programs (PLILP'97)*, LNCS 1292, pp:191-206, England, 1997.

[4] Frühwirth T., Theory and practice of constraint handling rules. In *The Journal of Logic Programming*, 37, pp:95-138, 1998.

[5] Van Hentenryck P., Tutorial on the CHIP systems and applications. In *Workshop of Constraint Logic Programming*, Rehovot, Israel, Weizmann Institute of Science, 1988.

[6] Van Hentenryck P., *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.