Entailment of Finite Domain Constraints

Björn Carlson and **Mats Carlsson**
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden
{bjornc,matsc}@sics.se

Daniel Diaz
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay, France
Daniel.Diaz@inria.fr

Abstract Using a glass-box theory of finite domain constraints, FD, we show how the entailment of user-defined constraints can be expressed by anti-monotone FD constraints. We also provide an algorithm for checking the entailment and consistency of FD constraints. FD is shown to be expressive enough to allow the definition of arithmetical constraints, as well as non-trivial symbolic constraints, that are normally built in to CLP systems. In particular, we use conditional FD constraints, which exploit entailment checking, to define symbolic constraints. Thus, we claim that a glass-box system such as FD is expressive enough to capture the essence of finite domain constraint programming.

Keywords: constraint logic programming, finite domain constraints, entailment, monotonicity.

1 Introduction

The glass-box approach to constraint logic programming consists in controlling the constraint solver at a more detailed level than what is possible in a system where the solver is provided as a black box. Constraints that are builtins of a black-box solver, are instead defined by programming primitives of the glass-box [5, 13, 4, 1]. Combinators of a glass-box system typically include conjunction, implication and disjunction.

The benefits of using glass-box systems are that the programmer is given *more freedom* of how to specify a problem, since the constraints can be tailored with respect to the problem at hand, and that the problem can be solved *more efficiently* since it need not be reduced to fit the constraints of the solver.

Furthermore, the implementation of a glass-box system lifts the complexity from the emulator level to the compiler level, as well as making available traditional compiler optimization techniques. The net result being that the implementations of glass-box systems can be made highly efficient [3, 13, 1].

In this paper we study how to use the glass-box system FD [12] to define

non-trivial finite domain constraints and to check their entailment. We show that by using conditional reasoning, based on entailment, complex symbolic constraints can be defined in FD. Entailment has previously been recognized as the key to how concurrent and constraint programming can be merged [8, 10], but hence it serves an important function even within a constraint logic programming framework.

By exploiting the monotonicity of FD constraints we show that entailment checking can be done purely in terms of anti-monotone FD constraints. Hence, FD is expressive enough both to define complex constraints and to check their entailment.

The method we propose works by translating a constraint definition into a sufficient truth condition, where some attention is put into making the condition minimal. The condition is translated into an anti-monotone FD constraint, which can efficiently be checked by an algorithm we provide.

The paper is structured as follows: we begin by describing the FD theory (Section 2), including a section on how to define constraints in FD, where some non-trivial constraints are defined. We then show how a constraint definition is translated into a sufficient truth condition (Section 3). A section follows which describes a translation of the truth conditions into anti-monotone FD constraints, and which includes algorithms for checking the monotonicity and the entailment of FD constraints (Section 4). A short summary concludes the paper (Section 5).

2 FD: A Theory of Finite Domain Constraints

The constraint system FD [12] is a general purpose constraint framework for solving discrete constraint satisfaction problems in a concurrent constraint setting. The theory is based on unary constraints by which higher arity constraints are defined, so for example constraints such as $X = Y$ or $X \leq 2Y$ are *defined* by FD constraints, instead of being built in to the theory. The unary constraints of FD are thought of as propagation rules, i.e. rules for describing node and arc consistency propagation.

The unary constraints of FD can be used as the target language for compilers of arbitrary finite domain constraints [2], and in fact FD subsumes basically all existing finite domain constraint systems with preserved and sometimes improved efficiency [1, 3, 4].

2.1 The theory

FD is based on *domain constraints* $X \in I$, where I is a set of integers described by a finite union of intervals. The set I is the set of *possible values* of X, and X is said to be *constrained* to I. $X \in I$ is satisfied by assigning a value in I to X.

A set S of domain constraints, where any two domain constraints $X \in I$ and $X \in I'$ have been replaced by $X \in (I \cap I')$, is called a *store*. Hence,

$$
\begin{array}{lll}
N & ::= & X \mid i, \text{ where } i \in \mathcal{Z} \\
T & ::= & N \mid T + T \mid T - T \mid T * T \mid T/T \mid T \textbf{ mod } T \mid \\
 & & \textbf{min}(R) \mid \textbf{max}(R) \\
R & ::= & T..T \mid T.. \mid ..T \mid R \,\&\, R \mid R : R \mid -R \mid \\
 & & R + T \mid R - T \mid R \textbf{ mod } T \mid \\
 & & \textbf{dom}(X) \\
C & ::= & N \textbf{ in } R \mid C \rightarrow C \mid C \wedge C
\end{array}
$$

Figure 1: Syntax of constraints in FD

a store S is *consistent* if there is no domain constraint $X \in \emptyset$ in S. The set which X is constrained to in S is denoted by X_S in the following. X is *determined* in S if $X_S = \{n\}$.

Suppose S_1 and S_2 are two stores. Let $S_1 \sqsubseteq S_2$ if for any variable X it holds $X_{S_2} \subseteq X_{S_1}$.

The computational primitive of FD, X **in** r, is a partial function from stores to domain constraints, such that X **in** r applied to a store S evaluates to a domain constraint $X \in r_S$, where r_S is the value of r in S (see below). The expression r is called a *range* (defined by R in Figure 1), which denotes a partial function from stores to finite unions of intervals over the integers. We will refer to X **in** r as an *indexical* in the following [12].

The partial function r is evaluated in a store S as follows. Any variable V not occurring as $\textbf{dom}(V)$ in r must be determined in S, and any variable V occurring as $\textbf{dom}(V)$ in r must be constrained in S to make r in S well-defined. The *value of a range* r in S, r_S, is thus *a set of integers* defined as: the expression $\textbf{dom}(Y)$ evaluates to Y_S, the expression $t_1..t_2$ is interpreted as the set $\{i \in \mathcal{Z} : t_{1_S} \leq i \leq t_{2_S}\}$, the expression $t..$ is interpreted as the set $\{i \in \mathcal{Z} : t_S \leq i\}$, the expression $..t$ is interpreted as the set $\{i \in \mathcal{Z} : i \leq t_S\}$, the operators : and & denote union and intersection respectively, the expressions $r + t$, $r - t$, and r **mod** t denote pointwise integer addition, subtraction, and modulo of r_S and t_S, where t cannot contain **max** or **min** terms, and finally the value of $-r$ in S is the set $\mathcal{Z} \backslash r_S$.

The *value of a term* t in S, t_S, is an *integer* defined as: a number is interpreted as itself, a variable is evaluated to its assignment, the interpretation of the arithmetical operators is the interpretation over the integers, and the expressions $\textbf{min}(r)$ and $\textbf{max}(r)$ evaluate to the infimum and supremum values of r_S. It is required that in a modulo expression t **mod** t_0, t_0 does not contain **max** or **min** terms. If every variable in t is determined in S, t is *determined* in S.

The set of FD constraints is the set of indexicals closed under (intuitionistic) implication and conjunction. In the following we sometimes refer to implications as *conditional constraints*.

Let S be a store, and c (d) be a constraint in FD. Thus define:

- S *entails* X **in** r if r is defined in S and $X_{S'} \subseteq r_{S'}$, for any S' such

that $S \sqsubseteq S'$.

- S entails $c \wedge d$ if S entails c and S entails d.
- S entails $c \rightarrow d$ if for every S', $S \sqsubseteq S'$, if S' entails c then S' entails d.
- c is *consistent* in S if for some S', $S \sqsubseteq S'$, S' entails c.
- c is *inconsistent* in S if c is not consistent in S.

Finally, r is *monotone* if for every pair of stores S_1 and S_2 such that $S_1 \sqsubseteq S_2$, $r_{S_2} \subseteq r_{S_1}$. r is *anti-monotone* if for every pair of stores S_1 and S_2 such that $S_1 \sqsubseteq S_2$, $r_{S_1} \subseteq r_{S_2}$.

X **in** r is monotone (anti-monotone) if r is monotone (anti-monotone), $c \wedge d$ is monotone (anti-monotone) if c and d are monotone (anti-monotone), and $c \rightarrow d$ is monotone (anti-monotone) if d is monotone (anti-monotone). Monotone constraints are used for adding domain constraints to the store, and anti-monotone constraints are used for checking entailment (see Section 3).

In the following we will use $\mathbf{min}(X)$ as shorthand for $\mathbf{min}(\mathbf{dom}(X))$. Also we use the variable r for ranges, the variable t for terms, the variables n for natural numbers and i for integers, and the variable c for constraints, where all the symbols may be indexed.

2.2 Defining constraints in FD

We define n-ary (arithmetical) constraints as FD constraints, the intention being that the denotation of the n-ary constraint should be captured by the interpretation of the FD expression.

Put more formally: suppose p is an n-ary relation over the integers, let H be $p(X_1, \ldots, X_n)$, and let S be a store. Then $S \Rightarrow H$ if $\{\langle a_1, \ldots, a_n \rangle : a_i \in X_{iS}\} \subseteq p$. Furthermore, suppose c is a constraint in FD. We then require the following to make c a definition of p.

1. If S entails c, then $S \Rightarrow H$.
2. If S determines X_i and $S \Rightarrow H$, then S entails c, $1 \leq i \leq n$.

In the following we adapt to the clause syntax of Prolog and use ":-" for definitions and "," for conjunction.

Example 1. The constraint $X = Y + 1$ can be defined as:

$X = Y + 1$:-
X **in** $\mathbf{dom}(Y) + 1$,
Y **in** $\mathbf{dom}(X) - 1$.

or as

$X = Y + 1$:-
X **in** $(\mathbf{min}(Y)+1)..(\mathbf{max}(Y)+1)$,
Y **in** $(\mathbf{min}(X)-1)..(\mathbf{max}(X)-1)$.

Hence, X (Y) is constrained by either the domain of Y (X), or by the minimum and maximum of Y (X). The domain approximation performs stronger propagation than the interval approximation, but the interval version is more efficient to compute. For a careful examination of domain and interval approximations of constraints see elsewhere [6, 13, 14].

Note that operationally the constraint propagation implemented by the FD constraints may be weaker than what can be performed by a constraint solver for the n-ary constraint.

Example 2. The constraint $X \neq Y$ can be defined either as

$X \neq Y$:-
    X **in** $-$ **dom**(Y),
    Y **in** $-$ **dom**(X).

or as

$X \neq Y$:-
    X **in** $-$ (**max**(Y)..**min**(Y)),
    Y **in** $-$ (**max**(X)..**min**(X)).

thus either the constraint is defined to be anti-monotone or monotone (see Section 4.3).

Example 3. We define the constraint $(X = A) \equiv B$, where B is 0 iff $X \neq A$ is true, and B is 1 iff $X = A$ is true, as

$(X = A) \equiv B$:-
    $X = A \rightarrow B = 1$,
    $B = 1 \rightarrow X = A$,
    $X \neq A \rightarrow B = 0$,
    $B = 0 \rightarrow X \neq A$.

Let us detail some symbolic constraints which need entailment detection, constraints that are normally built in to a CLP system but which can naturally be defined using conditionals.

Example 4. The magic series problem [11] consists in finding a sequence of numbers $\{X_0, \ldots, X_n\}$ such that i occurs X_i times in the sequence. The original formulation [11] used a `freeze` on each X_i. However, owing to entailment detection it is possible to simply encode the following relation [9]:

$$X_j = \sum_{i=0}^{n} (j)_i$$

where $(j)_i$ is 1 if $X_i = j$ and 0 if $X_i \neq j$. This is achieved by defining B_{ji} as $(X_i = j) \equiv B_{ji}$, and adding the constraints $\sum_{i=0}^{n} B_{ji} = X_j$, for each j between 0 and n.

Obviously, the constraint propagation obtained with this constraint is stronger than the propagation of the formulation using `freeze`. In [4] it is shown that the speedup with respect to the CHIP definition grows with n.

Finally, note that the constraint `atmost(N, L, V)`, which is true iff V occurs at most N times in L, can be defined by $\sum_{i=1}^{n} B_i \leq N$, where $L = [X_1, \ldots, X_n]$, and $(X_i = V) \equiv B_i$, $1 \leq i \leq n$.

Example 5. Similarly, the constraint `element(I,E,V)`, which holds iff the Ith element of E equals V, can be defined by conditional constraints as: let E be the list $[E_1, \ldots, E_k]$, and suppose $1 \leq I \leq k$ and B_i **in** $0 : i$ hold, for each i between 1 and k. The following constraints define the `element/3` relation:

- $I = i \rightarrow E_i = V$
- I **in** $\mathbf{dom}(B_1) : \cdots : \mathbf{dom}(B_k)$, where
- $E_i \neq V \rightarrow B_i = 0$, and
- $E_i = V \rightarrow B_i = i$.

However, note that this version of `element/3` does not exploit the full pruning possible from the denotation of the constraint [1].

3 Entailment Conditions

In this section we characterize the entailment of FD constraints by sufficient truth conditions.

3.1 Entailment of indexicals

The aim of this section is to show how to generate *logical* conditions to detect entailment and inconsistency of indexicals. The basic idea being that ranges are approximated by intervals, and thus entailment detection is made by reasoning over intervals. In later sections (Section 4 and 4.2) we use the conditions to generate anti-monotone indexicals which decide the conditions.

The general problem of deciding entailment of finite domain constraints belongs to NP, and thus we cannot expect to have efficient and complete entailment detection. Instead we choose to use entailment conditions which are efficient to compute (see Section 4.2), and in practice sufficiently strong.

In the following we consider only linear FD terms, which simplifies the presentation, but the tables can be generalized to hold for all FD terms.

Let *inf* (*sup*) be a function from linear terms to values which increases (decreases) as the computation progresses. That is, $inf(t)$ ($sup(t)$) is the smallest (largest) value that t can ever get (see Table 1).

Let m_k be the partial function such that $m_k(t) = t$ **mod** k when t is determined, let a_i be the function such that $a_i(t) = t + i$, let $f \circ g$ be defined as $(f \circ g)(t) = f(g(t))$, and let c be X **in** r, for some X and r. Let E_c (entailment condition) and D_c (inconsistency condition) be the two

t	$inf(t)$	$sup(t)$
i	i	i
$t_1 + t_2$	$inf(t_1) + inf(t_2)$	$sup(t_1) + sup(t_2)$
$t * n$	$inf(t) * n$	$sup(t) * n$
$t * (-n)$	$sup(t) * (-n)$	$inf(t) * (-n)$
$t_1 - t_2$	$inf(t_1) - sup(t_2)$	$sup(t_1) - inf(t_2)$
t/n	$inf(t)/n$	$sup(t)/n$
$t/(-n)$	$sup(t)/(-n)$	$inf(t)/(-n)$
i **mod** n	i **mod** n	i **mod** n
$\mathbf{min}(X)$	$\mathbf{min}(X)$	$\mathbf{max}(X)$
$\mathbf{max}(X)$	$\mathbf{min}(X)$	$\mathbf{max}(X)$

Table 1: Upper and lower bounds of linear FD-terms

r	$E(X, r, f)$
$t..$	$\mathbf{min}(X) \geq sup(f(t))$
$..t$	$\mathbf{max}(X) \leq inf(f(t))$
$t_1..t_2$	$\mathbf{min}(X) \geq sup(f(t_1)) \wedge \mathbf{max}(X) \leq inf(f(t_2))$
$\mathbf{dom}(Y)$	$E(X, \mathbf{min}(Y)..\mathbf{max}(Y), f)$
$r_1 : r_2$	$E(X, r_1, f) \vee E(X, r_2, f)$
$r_1 \ \& \ r_2$	$E(X, r_1, f) \wedge E(X, r_2, f)$
$-r$	$D(X, r, f)$
$r + t$	$E(X, r, f \circ a_t)$
$r - t$	$E(X, r, f \circ a_{-t})$
r **mod** t	$E(X, r, f \circ m_t)$

Table 2: Definition of $E(X, r, f)$

logical expressions defined by $E(X, r, a_0)$ and $D(X, r, a_0)$, where E and D are defined in Table 2 and Table 3.

Observe that if E_c (resp. D_c) is true in a store S then E_c (resp. D_c) is true in any store logically stronger than S. This follows from an inductive reasoning over the structure of c.

The *correctness* of the translation is shown by proving that if E_c (D_c) is true then c is entailed (inconsistent). This is done by induction over the structure of c. Thus, E_c and D_c are *sufficient* truth conditions for c.

Example 6. Let us consider the constraint (c) X **in** $\mathbf{min}(Y)..$, which can be used to impose $X \geq Y$. It follows from the above that $E_c \equiv \mathbf{min}(X) \geq \mathbf{max}(Y)$ and $D_c \equiv \mathbf{max}(X) < \mathbf{min}(Y)$. Thus, E_c is true (i.e. $X \geq Y$ is detected) as soon as all possible values of X are greater or equal to any possible value of Y, and D_c is true as soon as no possible value of X can be greater or equal to any possible value of Y.

r	$D(X, r, f)$
$t..$	$\mathbf{max}(X) < inf(f(t))$
$..t$	$\mathbf{min}(X) > sup(f(t))$
$t_1..t_2$	$\mathbf{max}(X) < inf(f(t_1)) \vee$
	$\mathbf{min}(X) > sup(f(t_2)) \vee$
	$inf(f(t_1)) > sup(f(t_2))$
$\mathbf{dom}(Y)$	$D(X, \mathbf{min}(Y)..\mathbf{max}(Y), f)$
$r_1 : r_2$	$D(X, r_1, f) \wedge D(X, r_2, f)$
$r_1 \ \& \ r_2$	$D(X, r_1, f) \vee D(X, r_2, f)$
$-r$	$E(X, r, f)$
$r + t$	$D(X, r, f \circ a_t)$
$r - t$	$D(X, r, f \circ a_{-t})$
$r \ \mathbf{mod} \ t$	$D(X, r, f \circ m_t)$

Table 3: DEFINITION OF $D(X, r, f)$

3.2 Entailment of user defined constraints

To deal with user defined constraints we need to generalize the truth conditions (see Section 3.1) to FD constraints as follows. Let c be an FD constraint. Then E_c (D_c) is defined as

- $E_{X \ \mathbf{in} \ r} = E(X, r, a_0)$ and $D_{X \ \mathbf{in} \ r} = D(X, r, a_0)$.
- $E_{c \wedge d} = E_c \wedge E_d$ and $D_{c \wedge d} = D_c \vee D_d$.
- $E_{c \to d} = D_c \vee E_d$ and $D_{c \to d} = E_c \wedge D_d$.

However, in many typical definitions $E_{c \wedge d}$ ($D_{c \wedge d}$) can be reduced to E_c (D_c) since $E_c \equiv E_d$ ($D_c \equiv D_d$).

Example 7. Let us consider the user constraint $X \geq Y$ defined as:

$X \geq Y$ $:-$
(c_X) X **in** $\mathbf{min}(Y)..$,
(c_Y) Y **in** $..\mathbf{max}(X)$.

From the above constraints it follows:

- $E_{c_X} \equiv E_{c_Y} \equiv \mathbf{min}(X) \geq \mathbf{max}(Y)$
- $D_{c_X} \equiv D_{c_Y} \equiv \mathbf{max}(X) < \mathbf{min}(Y)$

Hence, $E_{X \geq Y} = E_{c_X} \wedge E_{c_Y} \equiv E_{c_X}$, and $D_{X \geq Y} \equiv D_{c_X}$.

Example 8. Consider $X \neq Y$ defined as:

$X \neq Y$ $:-$
(c_X) X **in** $-\mathbf{dom}(Y)$,
(c_Y) Y **in** $-\mathbf{dom}(X)$.

The conditions detect when the domains of X and Y do *not overlap* anymore as:

$$
\begin{aligned}
E_{X \neq Y} &\equiv E_X \textbf{ in } {-}\textbf{dom}(Y) \text{ (c_X and c_Y are equivalent)} \\
&\equiv D_X \textbf{ in } \textbf{dom}(Y) \\
&\equiv D_X \textbf{ in } \textbf{min}(Y)..\textbf{max}(Y) \\
&\equiv \textbf{max}(X) < \textbf{min}(Y) \vee \textbf{min}(X) > \textbf{max}(Y)
\end{aligned}
$$

For checking the equivalence of two entailment conditions a normalization procedure can be used. Let c be a (monotone) FD constraint, and Π_c its associated truth (E_c) or falsity (D_c) condition.

The *normalization* of Π_c is done by rewriting Π_c into a disjunctive normal form, where each term in Π_c is replaced by its additive normal form. The rewriting is done by applying rewrite rules defined below. A rewrite rule applies if its template matches a substrate expression, modulo associativity and commutativity for $\{:, \&, *, +, \wedge, \vee\}$, replacing the substrate expression by a rewritten expression.

Given the following list of rewrite rules we iterate in top-down order through the list, applying a rule if it applies anywhere in the condition. If a rule is applied, the iteration is restarted from the beginning of the list. When no rule in the list applies, the iteration is terminated.

DNF. The *disjunctive normal form* of a condition is computed by the following rule:

$$E \wedge (E_1 \vee E_2) \Rightarrow (E \wedge E_1) \vee (E \wedge E_2)$$

ANF. The following rules compute the *additive normal form* of a term:

$$t * (t_1 \cdot t_2) \Rightarrow t * t_1 \cdot t * t_2 \ (\cdot \in \{+, -\})$$

MS. Subtraction is moved across inequalities as:

$$t_1 - t \cdot t_2 \Rightarrow t_1 \cdot t_2 + t, \cdot \in \{\leq, \geq\}$$
$$t_1 \cdot t_2 - t \Rightarrow t_1 + t \cdot t_2, \cdot \in \{\leq, \geq\}$$

Let Π_{c_1} and Π_{c_2} be two normalized entailment conditions. Π_{c_1} and Π_{c_2} are *equal* up to commutativity and associativity of $\wedge$ and $\vee$, if each corresponding pair of inequalities in Π_{c_1} and Π_{c_2} are equal. Two inequalities $t_1 \leq t_2$ and $t_3 \leq t_4$ are equal iff t_1 equals t_3 and t_2 equals t_4, where equality between terms is defined as identity up to commutativity and associativity of $+$ and $*$.

The *correctness* of the algorithm is shown by proving that if Π_{c_1} and Π_{c_2} are decided equivalent then Π_{c_1} is true iff Π_{c_2} is true. This is done by proving the correctness of each rewrite rule, and that the equivalence relation defined for normalized conditions is true equivalence.

The normalization *terminates* since **DNF** replaces a conjunction with two smaller conjunctions, **ANF** replaces a product with two smaller products, and the **MS** rules decrease the number of subtractions each time applied.

X **in** r in S	r monotone	r anti-monotone
$X_S \cap r_S = \emptyset$	inconsistent	may become entailed
$X_S \subseteq r_S$	may become inconsistent	entailed
$X_S \neq (X_S \cap r_S) \neq \emptyset$	may become inconsistent	may become entailed

Table 4: ENTAILMENT/CONSISTENCY OF X **in** r IN A STORE S

Note that the algorithm is *incomplete* since for example the two constraints X **in** 1..0 and X **in dom**(Y) & $-$ **dom**(Y) are logically equivalent, but are *not* decided such.

4 Entailment constraints

In this section we give a decision table for detecting entailment of indexicals, which is based on their monotonicity, and we show how to exploit this table to evaluate the entailment conditions (see Section 3.1). Furthermore we give an inductive definition of the monotonicity of X **in** r, which is needed to implement the entailment checking.

4.1 Entailment detection of X in r

Let c be X **in** r, and let S be the current constraint store. Suppose X is constrained in S to a (finite) set X_S, and let r_S be the value of r in S.

The entailment of c in S is checked using a case-analysis based on the value of X and r in S and on the monotonicity of r (see Table 4).

For example, suppose c is monotone and constrains X to the empty set in S. Then c is inconsistent in S since in any store stronger or equal to S, c will constrain X to the empty set. However, if c is anti-monotone and constrains X to the empty set in S, then there may be stores stronger than S in which c constrains X to something other than the empty set. Hence, c may or may not become entailed when S is strengthened.

If c is anti-monotone and X_S is a subset of r_S then c is entailed in S. Finally, if c is monotone and constrains X to something other than the empty set, c still may become inconsistent.

Computationally, whenever a constraint has become entailed it can be discarded, and whenever a constraint is inconsistent the computation fails. In all other cases the computation records (suspends) the constraint so that when the store is updated the constraint can be rechecked when necessary [1, 3]. If c is monotone, $X \in (X_S \cap r_S)$ is added to the store.

Example 9. Again we use disequality as an example. Suppose we define $X \neq Y$ as

$X \neq Y$:$-$
(c_X) $\quad X$ **in** $-$ **dom**(Y),
(c_Y) $\quad Y$ **in** $-$ **dom**(X).

r	$\nu_E(X, r, f)$
$t..$	$(sup(f(t)) - \mathbf{min}(X))..$
$..t$	$..(inf(f(t)) - \mathbf{max}(X))$
$t_1..t_2$	$(sup(f(t_1)) - \mathbf{min}(X))..(inf(f(t_2)) - \mathbf{max}(X))$
$\mathbf{dom}(Y)$	$\nu_E(X, \mathbf{min}(Y)..\mathbf{max}(Y), f)$
$r_1 : r_2$	$\nu_E(X, r_1, f) : \nu_E(X, r_2, f)$
$r_1 \,\&\, r_2$	$\nu_E(X, r_1, f) \,\&\, \nu_E(X, r_2, f)$
$-r$	$\nu_D(X, r, f)$
$r + t$	$\nu_E(X, r, f \circ a_t)$
$r - t$	$\nu_E(X, r, f \circ a_{-t})$
$r \ \mathbf{mod}\ t$	$\nu_E(X, r, f \circ m_t)$

Table 5: $E(X, r, f)$ EXPRESSED AS A RANGE

Observe that c_X and c_Y are anti-monotone. Suppose the domains of X and Y are disjoint in a given store S. Hence, $X_S \subseteq -Y_S$ and $Y_S \subseteq -X_S$, i.e. the entailment of c_X and c_Y in S is detected (see Table 4).

The decision table is *incomplete* since for example in the store $\{X \in \{1,2\}, Y \in \{3,4\}\}$ the monotone constraint X **in** $..\mathbf{max}(Y)$ is entailed without being detected such by Table 4. Only when Y is determined the constraint will be decided entailed (see Section 4.3).

This scheme has been implemented in the AKL-system, developed at SICS [7], and preliminary results indicate an efficiency comparable with `clp`(FD), `cc`(FD), and `CHIP` [1].

4.2 Generating entailment checking indexicals

In this section we show how to use the entailment detection in Section 4 for checking the entailment conditions of Section 3.1. We adapt Table 2 and Table 3 to generate anti-monotone indexicals instead of conditions, and thus we can use the decision table (Table 4) for checking the conditions [1].

Two operators are defined, ν_E and ν_D (see Table 5 and 6), such that the indexical 0 **in** $\nu_E(X, r, a_0)$ is entailed iff the condition $E(X, r, a_0)$ is true and the indexical 0 **in** $\nu_D(X, r, a_0)$ is entailed iff the condition $D(X, r, a_0)$ is true, which can be proven by induction over r.

Furthermore, $\nu_E(X, r, a_0)$ and $\nu_D(X, r, a_0)$ are anti-monotone, since $t..$, $..t$, and $t_1..t_2$ are mapped onto anti-monotone ranges and anti-monotonicity is preserved by unions and intersections. Thus, if $E(X, r, a_0)$ $(D(X, r, a_0))$ is true then 0 **in** $\nu_E(X, r, a_0)$ $(0$ **in** $\nu_D(X, r, a_0))$ is decided entailed by Table 4.

Note that 0 **in** $\nu_E(X, r, a_0)$ $(0$ **in** $\nu_D(X, r, a_0))$ contains all variables in the indexical X **in** r. So these conditions will be (re)tested each time a variable of X **in** r is modified until the constraint is true or false.

Finally, let ν_{Ec} (ν_{Dc}) correspond to $\nu_E(X, r, a_0)$ $(\nu_D(X, r, a_0))$ in the following, where c is X **in** r for some X and r. Thus, $E_c \wedge E_d$ is mapped

r	$\nu_D(X, r, f)$
$t..$	$..(inf(f(t)) - \mathbf{max}(X) - 1)$
$..t$	$..(\mathbf{min}(X) - sup(f(t)) - 1)$
$t_1..t_2$	$..(inf(f(t_1)) - \mathbf{max}(X) - 1) :$
	$..(\mathbf{min}(X) - sup(f(t_2)) - 1) :$
	$..(inf(f(t_1)) - sup(f(t_2)) - 1)$
$\mathbf{dom}(Y)$	$\nu_D(X, \mathbf{min}(Y)..\mathbf{max}(Y), f)$
$r_1 : r_2$	$\nu_D(X, r_1, f)$ & $\nu_D(X, r_2, f)$
r_1 & r_2	$\nu_D(X, r_1, f) : \nu_D(X, r_2, f)$
$-r$	$\nu_E(X, r, f)$
$r + t$	$\nu_D(X, r, f \circ a_t)$
$r - t$	$\nu_D(X, r, f \circ a_{-t})$
r **mod** t	$\nu_D(X, r, f \circ m_t)$

Table 6: $D(X, r, f)$ EXPRESSED AS A RANGE

onto 0 **in** ν_{Ec} & ν_{Ed}, $D_c \vee D_d$ is mapped onto 0 **in** $\nu_{Dc} : \nu_{Dd}$, $D_c \vee E_d$ is mapped onto 0 **in** $\nu_{Dc} : \nu_{Ed}$, and $E_c \wedge D_d$ is mapped onto 0 **in** ν_{Ec} & ν_{Dd}.

4.3 Computing monotonicity of X in r

We now give an inductive definition of the monotonicity of X **in** r which is used as the basis of an algorithm for checking the monotonicity. We use mutually recursive definitions (Table 7 and 8) to compute when r is monotone, and when r is anti-monotone. The definitions state which variables occurring in r that must be determined before r is monotone ($\mathcal{M}_r$), and which variables in r which must be determined before r is anti-monotone ($\mathcal{A}_r$).

Intuitively, the monotonicity of a range is preserved under set arithmetical operations, union, intersection, and inverted by the complement operator. The monotonicity of the interval combinator $t_1..t_2$ depends on whether the terms t_1 and t_2 are increasing or decreasing expressions. The increase/decrease property of terms is preserved under addition and multiplication, and inverted in the second argument of subtraction and division. If r is monotone, the expression $\mathbf{min}(r)$ is an increasing expression, and the expression $\mathbf{max}(r)$ is a decreasing expression. If r is anti-monotone, the expression $\mathbf{min}(r)$ is a decreasing expression, and the expression $\mathbf{max}(r)$ is an increasing expression.

In the following we consider only linear FD terms, which simplifies the presentation.

Let t be a linear term. The sets $\mathcal{S}_t$ (shrinking) and $\mathcal{G}_t$ (growing) are two sets of variables defined by Table 7. The intuition being that if all variables in $\mathcal{S}_t$ ($\mathcal{G}_t$) are determined (constants), then t takes on decreasing (increasing) values. If $\mathcal{S}_t$ and $\mathcal{G}_t$ both are empty, t denotes a unique natural number.

Let r be a range. The sets $\mathcal{M}_r$ and $\mathcal{A}_r$ are defined by Table 8. The

t	$\mathcal{S}_t$	$\mathcal{G}_t$
n	$\emptyset$	$\emptyset$
$t_1 + t_2$	$\mathcal{S}_{t_1} \cup \mathcal{S}_{t_2}$	$\mathcal{G}_{t_1} \cup \mathcal{G}_{t_2}$
$t * n$	$\mathcal{S}_t$	$\mathcal{G}_t$
$t * (-n)$	$\mathcal{G}_t$	$\mathcal{S}_t$
$t_1 - t_2$	$\mathcal{S}_{t_1} \cup \mathcal{G}_{t_2}$	$\mathcal{G}_{t_1} \cup \mathcal{S}_{t_2}$
t/n	$\mathcal{S}_t$	$\mathcal{G}_t$
$t/(-n)$	$\mathcal{G}_t$	$\mathcal{S}_t$
t_1 **mod** t_2	$\mathcal{S}_{t_1} \cup \mathcal{S}_{t_2} \cup \mathcal{G}_{t_2}$	$\mathcal{G}_{t_1} \cup \mathcal{S}_{t_2} \cup \mathcal{G}_{t_2}$
min(r)	$\mathcal{A}_r$	$\mathcal{M}_r$
max(r)	$\mathcal{M}_r$	$\mathcal{A}_r$

Table 7: MONOTONICITY OF LINEAR TERMS

r	$\mathcal{M}_r$	$\mathcal{A}_r$
$t..$	$\mathcal{G}_t$	$\mathcal{S}_t$
$..t$	$\mathcal{S}_t$	$\mathcal{G}_t$
$t_1..t_2$	$\mathcal{G}_{t_1} \cup \mathcal{S}_{t_2}$	$\mathcal{S}_{t_1} \cup \mathcal{G}_{t_2}$
dom(V)	$\emptyset$	$\{V\}$
$r_0 \cdot t \quad (\cdot \in \{+, -, \mathbf{mod}\})$	$\mathcal{M}_{r_0} \cup \mathcal{S}_t \cup \mathcal{G}_t$	$\mathcal{A}_{r_0} \cup \mathcal{S}_t \cup \mathcal{G}_t$
$r_1 \cdot r_2 \quad (\cdot \in \{ :\ ,\ \& \})$	$\mathcal{M}_{r_1} \cup \mathcal{M}_{r_2}$	$\mathcal{A}_{r_1} \cup \mathcal{A}_{r_2}$
$-r_0$	$\mathcal{A}_{r_0}$	$\mathcal{M}_{r_0}$

Table 8: MONOTONICITY OF RANGES

intuition being that if all variables in $\mathcal{M}_r$ ($\mathcal{A}_r$) are determined (constants), then r denotes a monotone (anti-monotone) range. If $\mathcal{M}_r$ and $\mathcal{A}_r$ both are empty, r denotes a unique set.

Example 10.

- Let $r = 1..3$. Then $\mathcal{M}_r = \mathcal{A}_r = \emptyset$.
- Let $r = \mathbf{dom}(Y)$. Then $\mathcal{M}_r = \emptyset$ and $\mathcal{A}_r = \{Y\}$.
- Let $r = \mathbf{dom}(Y) :\ -\mathbf{dom}(Z)$. Then $\mathcal{M}_r = \{Z\}$ and $\mathcal{A}_r = \{Y\}$.

Hence, X **in** r is monotone if all variables in $\mathcal{M}_r$ are determined and anti-monotone if all variables in $\mathcal{A}_r$ are determined. The *complexity* of checking the monotonicity is the complexity of the union-procedure multiplied by the numbers of operators in r, i.e. basically $O(|r| v \log v)$, where v is the number of variables in r.

The *correctness* of the tables is shown by induction on r. Furthermore, by induction on r it can be proven that $\mathcal{A}_{\nu_E(X,r,a_0)} = \emptyset = \mathcal{A}_{\nu_D(X,r,a_0)}$. Hence, combining Table 8 with Table 4 gives an algorithm for checking the entailment conditions of section 3.

Observe that we do not have a complete decidability procedure for monotonicity. Ranges such as $\mathbf{dom}(Y) : -\mathbf{dom}(Y)$ and $\mathbf{dom}(Y)\ \&\ -\mathbf{dom}(Y)$ cannot be classified until Y is determined, even though they both denote a unique set in any store ($\mathcal{Z}$ and $\emptyset$ respectively).

5 Conclusion

In this paper we consider the entailment of finite domain constraints. Given a finite domain constraint c, defined by an FD constraint, an anti-monotone FD constraint is derived denoting a sufficient truth condition for c. We provide an efficient algorithm for checking the entailment of anti-monotone FD constraints.

Conditional finite domain constraints exploit entailment detection and are shown to be sufficient for defining some non-trivial symbolic constraints. Thus, this implies that many high-level constraints, builtins of existing CLP systems, can be user-defined in a system such as FD while still being efficient.

Current and future research concerns the entailment, compilation, and implementation of logical combinations of constraints, such as disjunctions and implications of finite domain constraints.

Acknowledgements: This work has partly been financed by ACCLAIM, ESPRIT Project 7195. We also owe Philippe Codognet at INRIA-Rocquencourt, Seif Haridi and Torkel Franzén at SICS many thanks for their assistance throughout the work. Finally, we would like to thank the anonymous referees for their helpful recommendations.

References

[1] B. Carlson, S. Janson and S. Haridi. Programming in AKL(FD). Forthcoming SICS Research Report, 1994.

[2] B. Carlson and M. Carlsson. Compiling Linear Integer Constraints. Forthcoming SICS Research Report, 1994.

[3] D. Diaz and P. Codognet. A Minimal Extension of the WAM for `clp(FD)`. In *Proceedings of the 10th International Conference on Logic Programming*, 1993.

[4] D. Diaz and P. Codognet. Compiling Constraint in `clp(FD)`. Technical report, INRIA-Rocquencourt, 1993.

[5] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988.

[6] M. Dincbas, P. van Hentenryck, and H. Simonis. Constraint Satisfaction using constraint logic programming. In *Artifical Intelligence*, vol 58, 113-159, 1992.

[7] S. Janson and S. Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, 1991.

[8] M. J. Maher. Logic semantics for a class of committed choice programs. In *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press, 1987.

[9] W.J. Older and F. Benhamou. Programming in CLP(BNR). In *Position Papers of the First Workshop, PPCP*, Newport, Rhode Island, 1993.

[10] V. A. Saraswat. *Concurrent Constraint Programming*, MIT Press, 1993.

[11] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[12] P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(FD). Unpublished manuscript, Computer Science Department, Brown University, 1991.

[13] P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint Logic Programming over Finite Domains: the Design, Implementation, and Applications of cc(FD). Technical report, Computer Science Department, Brown University, 1992.

[14] P. van Hentenryck and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. In *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, 1991.