An Interval Lattice-based Constraint Solving Framework for Lattices

Antonio J. Fernández[1]* and Patricia M. Hill[2]

[1] Departamento de Lenguajes y Ciencias de la Computación, E.T.S.I.I., 29071 Teatinos, Málaga, Spain email:afdez@lcc.uma.es
[2] School of Computer Studies,University of Leeds, Leeds, LS2 9JT, England email:hill@scs.leeds.ac.uk

Abstract. We present a simple generic framework to solve constraints on any domain (finite or infinite) which has a lattice structure. The approach is based on the use of a single constraint similar to the indexicals used by CLP over finite domains and on a particular definition of an interval lattice built from the computation domain. We provide the theoretical foundations for this framework, a schematic procedure for the operational semantics, and numerous examples illustrating how it can be used both over classical and new domains. We also show how lattice combinators can be used to generate new domains and hence new constraint solvers for these domains from existing domains.
Keywords: Lattice, constraint solving, constraint propagation, indexicals.

1 Introduction

Constraint Logic Programming (CLP) systems support many different domains such as finite ranges of integers, reals, finite sets of elements or the Booleans. The type of the domain determines the nature of the constraints and the solvers used to solve them. Existing constraint solvers (with the exception of the CHR approach [7]), only support specified domains. In particular, the cardinality of the domain determines the constraint solving procedure so that existing CLP systems have distinct constraint solving methods for the finite and the infinite domains. On the other hand, CHR [7] is very expressive, allowing for user-defined domains. Unfortunately this flexibility has a cost and CHR solvers have not been able to compete with the other solvers that employ the more traditional approach. In this paper we explore an alternative approach for a flexible constraint solver that allows for user and system defined domains with interaction between them.

Normally, for any given domain, a solver has many constraints, each with its own bespoke implementation. The exception to this rule is CLP(FD) [4] which is designed for the finite domain of integers and based on a single generic constraint

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

often referred to as an *indexical*. The implementation of indexicals uses a simple interval narrowing technique which can be smoothly integrated into the WAM [2, 6]. This approach has been shown to be adaptable and very efficient and now integrated into mainstream CLP systems such as SICStus Prolog.

This paper has two contributions. First, we provide a theoretical framework for the indexical approach to constraint solvers. This is formulated for any ordered domain that is a lattice. We have observed that most of the existing constraint solvers are for domains that are lattices. Thus our second contribution is to provide a theoretical foundation for more generic constraint solvers where a single solver can support any system or user-defined domain (even if its cardinality is infinite) provided it is a lattice. One advantage of our framework is that, as it is based on lattice theory, it is straightforward to construct new domains and new constraint solvers for these domains from existing ones. In this paper, we describe different ways of performing these constructions and illustrate them by means of examples.

The paper is structured as follows. Section 2 recalls algebraic concepts used in the paper. In Section 3 the computation domain, the execution model and a schema of an operational semantics are described. Section 4 shows the genericity of the theoretical framework by providing several instances which include both the common well-supported domains as well as new domains. Section 5 describes with examples how the framework can be used on the combination of domains. The paper ends with some considerations about related work and the conclusions.

2 Preliminaries

2.1 Ordered Sets

Definition 1. *(Ordering) Let C be a set with equality. A binary relation $\preceq$ on C is an* ordering *relation if it is reflexive, antisymmetric and transitive. The relation $\prec$ can be defined in terms of $\preceq$*

$$c \prec c' \Leftrightarrow c \preceq c' \wedge c \neq c',$$
$$c \preceq c' \Leftrightarrow c \prec c' \vee c = c'.$$

We write $c \preceq_C c'$ (when necessary) to express that $c \preceq c'$ where $c, c' \in C$. Let C be a set with ordering relation $\preceq$ and $c, c' \in C$. Then we write $c \sim c'$ if either $c \preceq c'$ or $c' \preceq c$ and $c \not\sim c'$ otherwise. Any set C which has an ordering relation is said to be ordered. *Evidently any subset of an ordered set is ordered.*

Definition 2. *(Dual of an ordered set) Given any ordered set C we can form a new ordered set $\hat{C}$ (called the* dual of *C) which contains the same elements as C and $b \preceq_{\hat{C}} a$ if and only if $a \preceq_C b$. In general, given any statement Φ about ordered sets, the* dual statement *$\hat{\Phi}$ may be obtained by replacing each expression of the form $x \preceq y$ by $y \preceq x$.*

Definition 3. *(Bounds) Let C be an ordered set. An element s in C is a* lower (upper) bound *of a subset $E \subseteq C$ if and only if $\forall x \in E$: $s \preceq x$ ($x \preceq s$). If the set* *of lower (upper) bounds of E has a greatest (least) element, then that element is called the* greatest lower bound *(*least upper bound*) of E and denoted by $glb_C(E)$ ($lub_C(E)$). For simplicity, we adopt the notation $glb_C(x, y)$ and $lub_C(x, y)$ when E contains only two elements x and y.*

Definition 4. *(Predecessor and successor) Let C be an ordered set and let $c, c' \in C$. Then c is called a* predecessor *of c' and c' a* successor *of c if $c \preceq c'$. We say c is the* immediate predecessor *of c' if $c \prec c'$ and for any $c'' \in C$ such that $c \preceq c'' \prec c'$ implies $c = c''$. The* immediate successor *of c is defined dually.*

Definition 5. *(Direct product) Let C_1 and C_2 be ordered sets. The* direct product *$C = \langle C_1, C_2\rangle$ is an ordered set with domain the Cartesian product of C_1 and C_2 and ordering defined by: $\langle x_1, x_2\rangle \preceq_C \langle y_1, y_2\rangle \iff x_1 \preceq_{C_1} y_1$ and $x_2 \preceq_{C_2} y_2$*

Definition 6. *(Lexicographic product) Let C_1 and C_2 be ordered sets. The* lexicographic product *$C = (C_1, C_2)$ is an ordered set with domain the Cartesian product of C_1 and C_2 and ordering defined by:*

$$(x_1, x_2) \preceq_C (y_1, y_2) \iff x_1 \prec_{C_1} y_1 \text{ or } x_1 = y_1 \text{ and } x_2 \preceq_{C_2} y_2$$

2.2 Lattices

Definition 7. *(Lattice) Let L be an ordered set. L is a* lattice *if $lub_L(x, y)$ and $glb_L(x, y)$ exist for any two elements $x, y \in L$. If $lub_L(S)$ and $glb_L(S)$ exist for all $S \subseteq L$, then L is a* complete lattice.

Definition 8. *(Top and bottom elements) Let L be a lattice. $glb_L(L)$, if it exists, is called* the bottom element *of L and written $\bot_L$. Similarly, $lub_L(L)$, if it exists, is called* the top element *of L and written $\top_L$. The lack of a bottom or top element can be remedied by adding a fictitious one. Thus, we define the* lifted lattice *of L to be $L \cup \{\bot_L, \top_L\}$ where, if $glb_L(L)$ does not exist, $\bot_L$ is a new element not in L such that $\forall a \in L, \bot_L \prec a$ and similarly, if $lub_L(L)$ does not exist, $\top_L$ is a new element not in L such that $\forall a \in L, a \prec \top_L$.*

Proposition 1. *(Products of lattices) Let L_1 and L_2 be two (lifted) lattices. Then the direct product $\langle L_1, L_2\rangle$ and the lexicographic product (L_1, L_2) are lattices when we define:*

$$\begin{aligned}
glb(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) &= \langle glb_{L_1}(x_1, y_1), glb_{L_2}(x_2, y_2)\rangle \\
glb((x_1, x_2), (y_1, y_2)) &= \mathit{if}\ x_1 = y_1\ \mathit{then}\ (x_1, glb_{L_2}(x_2, y_2)) \\
&\quad \mathit{elsif}\ x_1 \prec y_1\ \mathit{then}\ (x_1, x_2) \\
&\quad \mathit{elsif}\ x_1 \succ y_1\ \mathit{then}\ (y_1, y_2) \\
&\quad \mathit{else}\ (glb_{L_1}(x_1, y_1), \top_{L_2})
\end{aligned}$$

lub is defined dually to glb[1].

Proofs and more information about lattices can be found in [5].

[1] Note that $\top_{L_2}$ must be also changed to its dual $\bot_{L_2}$.

3 The Constraint Domains

3.1 The Computation Domain

The underlying domain for the constraints, denoted here by D_0, is a lattice called the *computation domain*. It is assumed that D_0 has been lifted to include top and bottom elements, $\top_{D_0}$ and $\bot_{D_0}$ respectively.

The domain that is actually used for the constraint solving is a set of intervals on the computation domain and called the *interval domain*. We allow for the bounds of the interval to be either open or closed and denote these bounds with open and closed brackets, respectively. Thus, we first need to define an ordering between the open and closed right brackets ')', ']' so that the domain of right brackets is itself a lattice.

Definition 9. *(Bracket domain) The* bracket domain B *is the lattice of two elements ')' and ']' with ordering ')'* $\prec_B$ *']'. Any element of* B *is denoted by '}'.*

Definition 10. *(Simple bounded computation domain) The* simple bounded computation domain D *is the lexicographic product* (D_0, B).

By Proposition 1, D is a lattice. For clarity we write a} to express (a,'}') in D. For example, if D_0 is the integer domain, then in $D = (D_0, B)$, $3) \preceq_D 3]$, $4] \preceq_D 7]$, $glb_D(3], 5]) = 3]$ and $lub_D(3], 3)) = 3]$. Note that $\bot_D = \bot_{D_0})$ and that $\top_D = \top_{D_0}]$.

Definition 11. *(Mirror of D) The* mirror of D *is the lexicographic product* $(\hat{D_0}, B)$ *and is denoted by* $\overline{D}$. *The mirror of an element* $t \in D$ *is denoted by* $\overline{t}$. *By Proposition 1,* $\overline{D}$ *is a lattice. For convenience, we write* $\{a$ *to express* $\overline{a\}}$.

Note that if $t_1 = a_1\}_1, t_2 = a_2\}_2 \in D$ where $a_1 \neq a_2$ we have:
(1) $\overline{t_2} \preceq_{\overline{D}} \overline{t_1} \Leftrightarrow t_1 \preceq_D t_2$;
(2) $glb_{\overline{D}}(\overline{t_1}, \overline{t_2}) = \overline{lub_D(t_1, t_2)}$ and $lub_{\overline{D}}(\overline{t_1}, \overline{t_2}) = \overline{glb_D(t_1, t_2)}$;
(3) $\bot_{\overline{D}} = \overline{\top_{D_0})} = (\top_{D_0}$, $\top_{\overline{D}} = \overline{\bot_{D_0}]} = [\bot_{D_0}$.

For example, if $D_0 = \Re$, $\overline{3.1]} = [3.1$ and $\overline{6.7)} = (6.7$, $[5.2 \preceq_{\overline{D}} (3.1 \preceq_{\overline{D}} [3.1 \preceq_{\overline{D}} [2.2$, $glb_{\overline{D}}([5.0, [7.2) = [7.2$ and $lub_{\overline{D}}([5.0, [7.2) = [5.0$.

3.2 Constraint Operators

Let $D = (D_0, B)$ be the simple bounded computation domain for D_0.

Definition 12. *(Constraint operators) A* constraint operator (for D) *is a function* $\circ :: D_1 \times D_2 \to D$ *where* $D_1, D_2 \in \{D, \overline{D}\}$. *Given a constraint operator* $\circ$, *the* mirror operator $\overline{\circ} :: \overline{D_1} \times \overline{D_2} \to \overline{D}$ *is defined, for each* $t_1 \in D_1$ *and* $t_2 \in D_2$, *to be* $\overline{t_1} \overline{\circ} \overline{t_2} = \overline{t_1 \circ t_2}$.

Definition 13. *(Monotonicity of operators) Suppose* $D_1, D_2 \in \{D, \overline{D}\}$ *and* $\circ :: D_1 \times D_2 \to D$ *is a constraint operator. Then,* $\circ$ *is* monotonic *if, for all* $t_1, t_1' \in D_1$ *and* $t_2, t_2' \in D_2$ *such that* $t_1 \preceq_{D_1} t_1'$ *and* $t_2 \preceq_{D_2} t_2'$ *we have*

$$(t_1 \circ t_2) \preceq_D (t_1' \circ t_2)$$
$$(t_1 \circ t_2) \preceq_D (t_1 \circ t_2').$$

Lemma 1. *The constraint operator $\circ$ is monotonic if and only if the mirror operator $\overline{\circ}$ is monotonic.*

We impose the following restriction on the constraint operators.

Law of monotonicity for constraint operators.

- Each constraint operator must be monotonic.

Normally, a constraint operator $\circ :: D_1 \times D_2 \to D$ where $D_1, D_2 \in \{D, \overline{D}\}$ will be defined by the user or system on D_0 and B separately. The value of $\circ$ on D is then inferred from its value on D_0 and B so that, if $t_1 = a_1\}_1, t_2 = a_2\}_2$ are terms in D_1, D_2, respectively, then $t_1 \circ t_2 \; = \; (a_1 \circ a_2)(\}_1 \circ \}_2)$. Then, if the law of monotonicity is to hold in D, it has to hold for the definitions of $\circ$ on each of D_0 and B.

For example, if $D_0 = \Re$, $3.0) + 4.0] = 7.0)$ where $)+] =)$ and $3.0 + 4.0 = 7.0$.

3.3 Indexicals

We now add indexicals to the domains D and $\overline{D}$. To distinguish between the simple bounded computation domain already defined and the same domain but augmented with indexicals, we denote the simple bounded computation domain for D_0 as D^s. We assume that there is both a set O_D of constraint operators defined on $D^s$and a set V_{D_0} of variables associated with the domain D_0.

Definition 14. *(Bounded computation domain) If D^s is a simple bounded computation domain for D_0, then the* bounded computation domain D for D_0 *and its mirror $\overline{D}$ are defined*

$$D = D^s \cup \{max(x) \mid x \in V_{D_0}\} \cup \{t_1 \circ t_2 \mid \circ :: D_1^s \times D_2^s \to D^s \in O_D, t_1 \in D_1, t_2 \in D_2\},$$
$$\overline{D} = \overline{D^s} \cup \{min(x) \mid x \in V_{D_0}\} \cup \{\overline{t_1 \circ t_2} \mid \circ :: D_1^s \times D_2^s \to D^s \in O_D, t_1 \in D_1, t_2 \in D_2\}.$$

where, if $t \in D \setminus D^s$

$$\overline{max(x)} = min(x),$$
$$\overline{t_1 \circ t_2} = \overline{t_1} \overline{\circ} \overline{t_2}.$$

The expressions $max(x), min(x)$ are called indexicals. *Elements of $D \setminus D^s$ and $\overline{D} \setminus \overline{D^s}$ are called* indexical terms

The bounded computation domain D is also a lattice inheriting its ordering from D^s. Thus, if $t_1, t_2 \in D$, then $t_1 \preceq_D t_2$ if and only if $t_1, t_2 \in D^s$ and $t_1 \preceq_{D^s} t_2$ or $t_1 = t_1' \circ t_1'', t_2 = t_2' \circ t_2''$ and $t_1' \preceq_D t_2', t_1'' \preceq_D t_2''$.

3.4 Interval Domain

Definition 15. *(Interval domain) We define the* interval domain R_D *over D_0 as the lattice resulting from the direct product $\langle \overline{D}, D \rangle$. The* simple interval domain R_D^s *is the lattice $\langle \overline{D^s}, D^s \rangle$.*

Therefore, for any $r_1 = \langle \overline{s_1}, t_1 \rangle$ *and* $r_2 = \langle \overline{s_2}, t_2 \rangle$, *where* $s_1, s_2, t_1, t_2 \in D$ *and* $r_1, r_2 \in R_D$,

$$\begin{aligned}
r_1 \preceq_{R_D} r_2 &\iff (\overline{s_1} \preceq_{\overline{D}} \overline{s_2}) \text{ and } (t_1 \preceq_D t_2),\\
glb_{R_D}(r_1, r_2) &= \langle glb_{\overline{D}}(\overline{s_1}, \overline{s_2}), glb_D(t_1, t_2)\rangle,\\
lub_{R_D}(r_1, r_2) &= \langle lub_{\overline{D}}(\overline{s_1}, \overline{s_2}), lub_D(t_1, t_2)\rangle,\\
\top_{R_D} &= [\bot_{D_0}, \top_{D_0}],\\
\bot_{R_D} &= (\top_{D_0}, \bot_{D_0}).
\end{aligned}$$

An element $\langle \overline{s}, t\rangle$ *in* R_D *is* inconsistent *if*
(1) $s \not\preceq_D t$ *(note that this means the range is inconsistent if* $s \not\prec_D t$*) or*
(2) $s = a)$ *and* $t = a\}$.
Otherwise $\langle \overline{s}, t\rangle$ *in* R_D *is* consistent. *Note that this means that* $\bot_{R_D}$ *is inconsistent.*

A range *is an element of* R_D^s. *A* range expression *is an element of* $R_D \setminus R_D^s$.

For simplicity, $\langle \overline{s}, t\rangle$ will be written as $\overline{s}, t$ for both ranges and range expressions. Thus an element $\langle \overline{a\}}, b\}\rangle$ is written as $\{a, b\}$. As examples of the definitions shown above and considering the real domain we have that $[2.3, 8.9)$ is a range, $[1.4, max(x) + 4.9]$ is a range expression, $[3.0, 4.0) \preceq_{R_D} (1.8, 4.5]$, $glb_{R_D}([3.2, 6.7], (1.8, 4.5]) = [3.2, 4.5]$ and $lub_{R_D}([3.2, 6.7], (1.8, 4.5]) = (1.8, 6.7]$. It is important to note that $\preceq_{R_D}$ simulates the interval inclusion.

3.5 Interval Constraints

Let R_D denote the interval domain over D_0 and let V_{D_0} be a set of variables associated with the domain D_0. An interval constraint for D_0 assigns an element in R_D to a variable in V_{D_0}.

Definition 16. *(Interval constraint) Suppose* $r \in R_D$ *and* $x \in V_{D_0}$. *Then*

$$x \sqsubseteq r$$

is called an interval constraint *for* D_0. x *is called the* constrained variable. *If* r *is a range (resp. range expression), then* $x \sqsubseteq r$ *is called a* simple *(resp.* non-simple*) interval constraint. The interval constraint* $x \sqsubseteq \top_{R_D}$ *is called a* type constraint *and denoted by* $x ::' D_0$. *A simple interval constraint* $x \sqsubseteq r$ *is* consistent *(resp.* inconsistent*) if* r *is consistent (resp. inconsistent).*

To illustrate these definitions: $y, x ::' Integer$, $b ::' Bool$, $w, t ::' Real$, and $n ::' Natural$ are examples of type constraints; $y \sqsubseteq [1, 4)$, $b \sqsubseteq [True, True]$, and $n \sqsubseteq [zero, suc(suc(zero))]$ are examples of simple interval constraints; $x \sqsubseteq min(y), max(y) + 3]$ and $t \sqsubseteq (1.21 \overline{*} min(w), 4.56)$ are examples of non-simple interval constraints where $+$ and $*$ are constraint operators for the $Integer$ and $Real$ domains.

Definition 17. *(A partial ordering on interval constraints) Suppose* $x \in V_{D_0}$. *Let* C_D^x *be the set of all interval constraints over* D_0 *with constrained variable*

x. *Suppose* $c_1 = x \sqsubseteq r_1, c_2 = x \sqsubseteq r_2 \in C_D^x$. *Then* $c_1 \preceq_{C_D^x} c_2$ *if and only if* $r_1 \preceq_{R_D} r_2$. *As* R_D *is a lattice,* C_D^x *is also a lattice. Note that* $glb_{C_D^x}(c_1, c_2) = x \sqsubseteq glb_{R_D}(r_1, r_2)$.

Definition 18. *(Constraint store) A* constraint store *for* D_0 *is a finite set of interval constraints for* D_0. *The set of all variables constrained in a store* S *is denoted by* X_S. *A constraint store* S *is in a* stable form *(or is* stable*)* wrt a set of variables $X \subseteq X_S$ *if for each* $x \in X$ *there is exactly one simple constraint* $x \sqsubseteq r$ *in* S. *If no set of variables is specified, we say that the store* S *is in a* stable form *if it is stable wrt* X_S. *A store is* inconsistent *if it contains at least one inconsistent interval constraint.*

Definition 19. *(Evaluating Indexical Terms) For each stable constraint store* S *for* D_0, *we define the (overloaded) evaluation functions*

$$eval_S :: D \to D^s, \qquad eval_S :: \overline{D} \to \overline{D^s}$$

$$\begin{aligned}
eval_S(t) &= t && \text{if } t \in D^s \cup \overline{D^s},\\
eval_S(max(x)) &= t && \text{if } x \sqsubseteq \overline{s}, t \in C^s,\\
eval_S(max(x)) &= \top_D && \text{if } C^s \text{ has no constraint for } x,\\
eval_S(min(x)) &= \overline{s} && \text{if } x \sqsubseteq \overline{s}, t \in C^s,\\
eval_S(min(x)) &= \bot_D && \text{if } C^s \text{ has no constraint for } x,\\
eval_S(t_1 \circ t_2) &= eval_S(t_1) \circ eval_S(t_2),
\end{aligned}$$

where C^s *is the set of all simple constraints in* S.

An indexical term is a generalisation of the indexical terms provided by CLP finite domain languages [4] and allow for infinite as well as finite ranges.

Remark 1. (Monotonicity of interval constraints) Note that with our definition of interval constraint we disallow a constraint such as[2] $x \sqsubseteq [10, 20] - max(y)$ by declaring the operator '$-$' as $- :: D \times \overline{D} \to D$ since $20] - max(y) \notin D$. This constraint is non-monotonic since, as the range of y decreases (so that $max(y)$ decreases), the term $20] - max(y)$ increases in D (so that the range of x increases). Observe that a range such as $[10, 20] - max(y)$ could not contribute to constraint propagation.

3.6 Execution Model

The execution model is based on a particular intersection of simple interval constraints and on two processes: the stabilisation of a constraint store and the constraint propagation.

[2] It is easier to understand this constraint when written as $x \sqsubseteq [10, -max(y) + 20]$.

Intersection of simple interval constraints

Definition 20. *($\cap_D$) The intersection in the domain D of two simple interval constraints $c_1 = x \sqsubseteq r_1$ and $c_2 = x \sqsubseteq r_2$ for the same* constrained variable *x is defined as follows:*

$$c_1 \ \cap_D \ c_2 = glb_{C_D}(c_1, c_2)$$

Note that this can be expressed in terms of ranges as follows:

$$(x \sqsubseteq r_1) \ \cap_D \ (x \sqsubseteq r_2) = x \sqsubseteq glb_{R_D}(r_1, r_2)$$

The following properties of $\cap_D$ are direct consequences of the definition.

Proposition 2. *($\cap_D$ Properties) Suppose $x \in V_{D_0}$ and c_1, c_2, c_3 are consistent constraints defined on the variable x where $c_3 = c_1 \cap_D c_2$. Then $\cap_D$ has the following properties:*

(1) Contractance: $c_3 \preceq_{C_D} c_1$ and $c_3 \preceq_{C_D} c_2$.

(2) Correctness: Only values which can't be part of any feasible solution, are removed. If $c \preceq_{C_D} c_1$ and $c \preceq_{C_D} c_2$, then $c \preceq_{C_D} c_3$.

(3) Commutativity: $(c_1 \ \cap_D \ c_2) = (c_2 \ \cap_D \ c_1)$

(4) Idempotence: The final constraint c_3 has to be computed once: $(c_1 \ \cap_D \ c_3) = c_3$ and $(c_3 \ \cap_D \ c_2) = c_3$.

If C^s is a set of simple constraints with the same constrained variable, then we define $\bigcap_D C^s = glb_{C_D}(C^s)$. As a result of the contractance property (1) in Proposition 2 we have $\cap_D C^s \preceq c^s$, for each $c^s \in C^s$.

Definition 21. *(Stabilised store) Let S be a constraint store and, for each $x \in X_S$, C^s_x the set of simple interval constraints constraining x in S. Then, the* stabilised store *S' of S is defined as follows:*

$$S' = (S \setminus \bigcup_{x \in X_S} C^s_x) \cup \{ \cap_D(C^s_x) \mid x \in X_S \}$$

Note that, by Definition 17, if $C^s_x = \emptyset$ then $\cap_{C_D}(C^s_x) = x \sqsubseteq \top_{R_D}$. This ensures that the stabilised store S' of S has exactly one simple interval constraint for each $x \in X_S$.

We write $S \mapsto S'$ to express that S' is the stabilised store of S.

Definition 22. *(Propagation of a constraint) Let c^{ns} be a non-simple interval constraint $x \sqsubseteq \overline{s}, t$ and S a stable constraint store. We say that c^{ns}* is propagated *(*using S*) to the simple interval constraint $x \sqsubseteq \overline{s_1}, t_1$ (denoted by c') if $eval_S(\overline{s}) = \overline{s_1}$ and $eval_S(t) = t_1$. We write $c^{ns} \leadsto^S c'$ to express that constraint c^{ns} has been propagated using S to c'.*

Definition 23. *(Store propagation) Let S be a stable store and C a set (possibly empty) of simple interval constraints. We say that S is propagated to C and write $S \leadsto C$ if $C = \{c \mid \exists c^{ns} \in S \wedge c^{ns} \leadsto^S c\}$.*

3.7 Operational Schema

In this section we present as a schema an outline procedure for the execution model. Let C be a set of interval constraints to be solved and let V be the set of all the variables constrained or indexed in C. Suppose $C = C^s \cup C^{ns}$ where C^s is the set of simple constraints in C and C^{ns} is the set of non-simple constraints in C.

Definition 24. *(Solution) A* solution for C *is a constraint store R that is stable with respect to V and containing only simple constraints where,*

$$(1)\ \forall c^s \in C^s\ \exists c \in R.c \preceq_{C_D} c^s,$$

$$(2)\ \forall c^{ns} \in C^{ns}\ \exists c \in R.c \preceq_{C_D}\ c^s, \textit{ where } c^{ns} \leadsto^{C'} c^s \textit{ and } C \mapsto C'.$$

We provide here a schema for computing a solution for C. Suppose $C \mapsto S$ and $S' = \emptyset$. The *operational schema* is as follows:

(1) *while* $S \neq S'$ *do*
(2) $S \leadsto C^s$ %% Constraint Propagation
(3) $S' := S$;
(4) $S' \cup C^s \mapsto S$; %% Store stabilisation
(5) *if* S is inconsistent then exit with fail *endif*
(6) *endwhile*

We do not discuss possible efficiency improvements here since the main aim here is to provide the basic methodology, showing how the execution method of CLP(FD) may be generalised for constraint solving on any domain with a lattice structure. If a solution exists, the solution is the set of all the simple interval constraints belonging to S.

Precision. New constraints, created by the propagation step (line 2), are added to the set of constraints before the stabilisation step (line 4). Thus, with infinite domains, the algorithm may not terminate (note that the constraints can be indefinitely contracted in the stabilisation step). To avoid it, we introduce the overloaded function *precision*/1 which is declared as $precision :: R^s_D \to \Re$. This function must satisfy the following properties:

(i) $precision(r) = 0.0$ if $r = \overline{s}, s$.
(ii) $precision(r_2) \leq precision(r_1)$ if $r_2 \preceq_{R_D} r_1$ (*Monotonicity*)

To allow for the lifted bounds for infinite domains, let Hr be the highest representable real in the computation machine. Then *precision* must also satisfy $precision(\top_{\overline{D}}, \top_D) = Hr$. The actual definition of *precision* depends on the computation domain. For Example:

- On the integer and $\Re$ domains: $precision(\{a, b\}) \Leftrightarrow \mid b–a \mid$.
- On the $\Re^2$ domain:

$$precision(\{(x_1, y_1), (x_2, y_2)\}) \Leftrightarrow \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

– On the set domain: $precision(\{s_1, s_2\}) \Leftrightarrow \#(s_2 \backslash s_1)$.

We overload $precision/1$ and define the precision of a simple interval constraint $c^s = x \sqsubseteq r$ as $precision(c^s) = precision(r)$ and the precision of a store S, which is stable wrt V, as $precision(S) = \sum_{x \in V, c^s \in S} precision(c^s)$.

By defining a computable[3] bound $\varepsilon \in \Re$, we can check if the precision of ranges for the simple constraints in a constraint store S were reduced by a significant amount in the stabilisation process. If the change is large enough then the propagation procedure continues. Otherwise the set of simple constraints in the store S is considered a "good enough" solution and the procedure terminates. The function $precision/1$ and bound ε are user or system defined for each computational domain.

To use $precision/1$ and ε, the operational schema needs to be extended with an extra test by replacing line (1) as follows:

(1) *while* $(S \neq S')$ and $(precision(S') - precision(S) \geq \varepsilon)$ *do*

As ranges in S and S' are contracted, $precision(S)$ and $precision(S')$ decrease by more than ε times the number of iterations of the loop *while*. Thus, there is a maximum number of possible iterations, depending on ε and the initial stabilised constraint store S.

Remark 2. (Some remarks on the precision map)

(1) A range can be contracted whereas its precision does not decrease (i.e. in the real domain, a range $r_1 = [-\infty, +\infty]$ can be contracted to a range $r_2 = [0, Hr]$ whereas $precision(r_1) = precision(r_2)$). To avoid an early termination, an additional test to check a change on the bounds of the ranges must also be added to the while loop condition.

(2) The bound ε allows a direct control over the accuracy of the results[4]. For example, $\varepsilon = 0.0$ for integers, $\varepsilon = 10^{-8}$ for reals and $\varepsilon = 0.0$ for sets. This provides the facility to obtain an *approximate solution* when an accurate solution may not be computable.

We show in the appendix that the extended operational schema has the following two properties.

1. *Termination.* The procedure shown above always terminates returning a fail or a solution.
2. *Correctness.* If it exists, the algorithm reaches a solution and this solution does not depend on the order in which constraints are chosen.

3.8 Improving Constraint Solving on Discrete Domains

We introduce two rules to improve our generic framework on discrete domains in which the *immediate predecessor* $pre(K)$ and *immediate successor* $suc(K)$ of

[3] That is, representable in the machine which is being used - the computation machine.

[4] [9] provided a similar idea but only over reals.

every value K in the domain can be computed. It is possible to eliminate the '(','')' brackets in favour of the '[',']' ones using the following two *range rules*:

$$\{a, K) \equiv \{a, pre(K)] \qquad \textbf{rleft}$$
$$(K, a\} \equiv [suc(K), a\} \qquad \textbf{rright}$$

If $\perp_D$ and $\top_D$ elements were added as fictitious bounds, we define: (1) $pre(\top_D) \equiv \top_D$ and (2) $suc(\perp_D) \equiv \perp_D$.

As an example, consider the Boolean domain with the ordering $false < true$ and the constraint $x \sqsubseteq [false, true)$. This constraint provides enough information to know the value of x must be $false$. Thus, given $suc(false) = true$ and $pre(true) = false$ and by applying **rleft**, the constraint $x \sqsubseteq [false, true)$ is transformed to $x \sqsubseteq [false, false]$.

As this domain is finite, the constraints could have been solved using an enumeration strategy[5] as is done in the existing finite domain constraint languages. However, by using immediate predecessors and successors, further constraint propagation may be generated without enumeration.

4 Instances of Our Framework

The framework can be used on many different domains. In this section, we present some examples. In the following, $(D_0, \preceq_{D_0}, glb_{D_0}, lub_{D_0}, \perp_{D_0}, \top_{D_0})$ denotes a lattice on D_0.

4.1 Classical Domains

Most classical constraint domains are lattices: $(Integer, \leq, min, max, -\infty, +\infty)$, $(\Re, \leq, min, max, -\infty, +\infty)$, $(Bool, \leq, \wedge, \vee, false, true)$ and $(Natural, \preceq, min, max, zero, \infty)$ are lattices under their usual orders and $false < true$. min and max functions return, respectively, the minimum and maximum element of any two elements in the computation domain. Here are examples of constraint intersection in the interval domain over these domains:

(1) $i \sqsubseteq [1, 8) \ \cap_D \ i \sqsubseteq (0, 5] \ = \ i \sqsubseteq [1, 5]$

(2) $r \sqsubseteq [1.12, 5.67) \ \cap_D \ r \sqsubseteq [2.34, 5.95) \ = \ r \sqsubseteq [2.34, 5.67)$

(3) $b \sqsubseteq (false, true] \ \cap_D \ b \sqsubseteq [false, true] \ = \ b \sqsubseteq (false, true]$

(4) $n \sqsubseteq [zero, suc(suc(zero))] \ \cap_D \ n \sqsubseteq [zero, suc(zero)] \ = \ n \sqsubseteq [zero, suc(zero)]$

4.2 Reasoning about Sets

$(Set\ D, \subseteq, \cap, \cup, \emptyset, \top_{Set\ D})$ is a lattice over which it is possible to solve set constraints. For example, consider $\{s ::' Set\ Integer, s \sqsubseteq [\{1\}, \{1, 2, 3, 4\}], s \sqsubseteq [\{3\}, \{1, 2, 3, 5\}]\}$ for solving. By applying $\cap_D$ twice, it is solved as follows:

$$s \sqsubseteq [\emptyset, \top_{Set\ Integer}] \ \cap_D \ s \sqsubseteq [\{1\}, \{1, 2, 3, 4\}] \ = \ s \sqsubseteq [\{1\}, \{1, 2, 3, 4\}]$$
$$s \sqsubseteq [\{1\}, \{1, 2, 3, 4\}] \ \cap_D \ s \sqsubseteq [\{3\}, \{1, 2, 3, 5\}] \ = \ s \sqsubseteq [\{1, 3\}, \{1, 2, 3\}]$$

[5] Possible values are assigned to the constrained variables and the constraints checked for consistency.

4.3 User Defined Domains

Binary Strings. The domain of binary strings $\sum^*$ is the set of all sequences (possibly infinite) of zeros and ones together with $\top_{\sum^*}$. The empty sequence is $\bot_{\sum^*}$. We define $x \preceq_{\sum^*} y$ if and only if x is a prefix (finite initial substring) of y. Note that, in the case, $x \not\preceq y$, $glb_{\sum^*}(x, y)$ is the largest common prefix of x and y (i.e.$glb_{\sum^*}(00010, 00111) = 00$, $glb_{\sum^*}(01, 00101) = 0$) and $lub_{\sum^*}(x, y)$ is $\top_{\sum^*}$. Then $(\sum^*, \preceq_{\sum^*}, glb_{\sum^*}, lub_{\sum^*}, \bot_{\sum^*}, \top_{\sum^*})$ is a lattice. This means is possible to define constraints on an interval lattice $\langle \overline{D}, D\rangle$ (with $D = \sum^* \times B$) i.e. $x, y ::' \sum^*, x \sqsubseteq [001\overline{+}min(y), \top_{\sum^*}]$ defines the interval of all strings which start with the substring 001. $+$ denotes the concatenation of strings.

Non Negative Integers Ordered by Division. Consider $(\mathcal{N}_d, \preceq_{\mathcal{N}_d})$ as the set of non negative integers (plus value 0) ordered by division, that is, for all $n, m \in \mathcal{N}_d$, $m \preceq_{\mathcal{N}_d} n$ iff $\exists k \in \mathcal{N}_d$ such that $km = n$ (that is, m divides n). This defines a partial order. Then any number) $(\mathcal{N}_d, \preceq_{\mathcal{N}_d}, gcd, lcm, 1, 0)$ is a lattice where gcd denotes the greatest common divisor function and lcm the least common multiple function. Thus our framework will solve constraints on this domain as follows: $x \sqsubseteq [2, 24] \ \cap_D x \sqsubseteq [3, 36] = x \sqsubseteq [6, 12]$.

Numeric Intervals We consider $Interv$ as the domain of the numeric intervals. We define $a \preceq_{Interv} b$ if and only if $a \subseteq b$. Thus glb_{Interv} and lub_{Interv} are the intersection and union of intervals respectively. Our framework solves constraints for the $Interv$ computational domain as follows:

$$i \sqsubseteq [[5, 6], [2, 10)] \ \cap_D \ i \sqsubseteq [(7, 9], [4, 15]] \ = \ i \sqsubseteq [[5, 6] \cup (7, 9], [4, 10)]$$

5 Combinations of Domains

Our lattice-based framework allows for new computation domains to be constructed from previously defined domains.

5.1 Product of Domains

As already observed, the direct and lexicographic products of lattices are lattices.

As an example, consider $\mathcal{N}_0 = \mathcal{N} \cup 0$ the domain of naturals plus 0. Then $\mathcal{N}_0$ is a lattice under the usual ordering. Note that $\bot_{\mathcal{N}_0} = 0$ and $\top_{\mathcal{N}_0}$ is lifted.
(1) Let $Point$ be the direct product domain $\mathcal{N}_0 \times \mathcal{N}_0$. Then, $Point$ is a lattice. Note that $\bot_{Point} = (0, 0)$ and $\top_{Point} = (\top_{\mathcal{N}_0}, \top_{\mathcal{N}_0})$.
(2) A rectangle can be defined by two points in a plane: its lower left corner and its upper right corner. Let $\Box$ be the direct product domain $Point \times Point$. Then, $\Box$ is a lattice. Note that $\bot_{\Box} = ((0, 0), (0, 0))$ and $\top_{\Box} = (\top_{Point}, \top_{Point})$

5.2 Sum of Domains

A lattice can be also constructed as a linear sum of other lattices.

Definition 25. *(Sum) Let $L_1, \ldots, L_n$ be lattices. Then their* linear sum $L_1 \oplus \ldots \oplus L_n$ *is the lattice L_S where:*
(1) $L_S = L_1 \cup \ldots \cup L_n$
(2) the ordering relation $\preceq_{L_S}$ is defined by:

$$x \preceq_{L_S} y \iff \begin{array}{l} x, y \in L_i \textit{ and } x \preceq_{L_i} y \\ \textit{or } x \in L_i,\ y \in L_j \textit{ and } i \prec j \end{array}$$

(3) glb_{L_S} and lub_{L_S} are defined as follows:

$$\begin{array}{l} glb_{L_S}(x,y) = glb_{L_i}(x,y) \textit{ and } lub_{L_S}(x,y) = lub_{L_i}(x,y) \textit{ if } x, y \in L_i \\ glb_{L_S}(x,y) = x \textit{ and } lub_{L_S}(x,y) = y \textit{ if } x \in L_i,\ y \in L_j \textit{ and } i \prec j \\ glb_{L_S}(x,y) = y \textit{ and } lub_{L_S}(x,y) = x \textit{ if } x \in L_i,\ y \in L_j \textit{ and } j \prec i \end{array}$$

and (4) $\bot_{L_S} = \bot_{L_1}$ and $\top_{L_S} = \top_{L_n}$

It is routine to check that the linear sum of lattices is a lattice. As an example, consider the lattice $AtoF$ containing all the (uppercase) alphabetic characters between 'A' and 'F' with the usual alphabetical ordering and $0to9$ the numeric characters from '$0'$ to '$9'$. Then the lattice of hexadecimal digits can be defined as the lattice $0to9 \oplus AtoF$.

6 Related Work

In addition to related work already discussed earlier in the paper, there are two other approaches to the provision of a general framework for constraint satisfaction. These are described in [3] and [1]. We discuss these here.

Bistarelli et al. [3] describe, for finite domains, a general framework based on a finite semiring structure (called c-semirings). They show that c-semirings can also be assimilated into finite complete lattices. This framework is shown to be adequate for classical domains and for domains which use a level of preference (i.e. cost or degree). However, unlike our proposal, they require the computational domain to be finite. Moreover, our framework does not require a level of confidence and, although they extended the approach of c-semirings to finite complete lattices and, in particular, for distributive lattices, they did not consider, as we have done, arbitrary lattices.

One important part of the definition of a constraint solver is the algorithm for constraint propagation and we have provided a simple schematic algorithm suitable for our constraint solving framework. In contrast, in [1], Apt focusses on just the algorithms and describes a generalisation for constraint propagation algorithms based on chaotic iterations. He shows how most of the constraint propagation algorithms presented in the literature can be expressed as instances of this general framework. Further work is needed to investigate the relationship between our algorithm and this framework.

7 Conclusions

In this paper we have defined a theoretical framework for constraint solving on domains with a lattice structure. Using such a domain, we have shown how to construct an interval lattice which allows the use of open, semi-open, semi-closed and closed intervals as well as infinite intervals. Variables, constraint operators and indexicals for each domain provide the tools for constructing interval constraints. We have shown that these constraints are a natural generalisation of the indexical constraints used in [4]. A schema for the operational semantics which is a modified form of the procedure proposed in [8] is also given and the main properties derived from it are studied. This schema is only partially specified making the incorporation of efficiency optimisations easier. To ensure termination, an idea from [9] for controlling accuracy in the processing of disjoint intervals over the reals has been generalised for our interval lattices.

Since the only requirement for our framework is that the computational domain must be a lattice, new domains can be obtained from previously defined domains using standard combinators (such as direct product and sum). We have provided examples to highlight the potential here.

To demonstrate the feasibility of our approach we have implemented a prototype (built using CHRs [7]). This is still being improved and extended but the latest version may be obtained from $http://www.lcc.uma.es/\sim afdez/generic$.

References

1. Apt K.R., From Chaotic Iteration to Constraint Propagation. In Proc. of the *24th International Colloquium on Automata, Languages and Programming (ICALP'97)* (invited lecture), LNCS 1256, pp:36-55, 1997.
2. Ait-kaci H., Warren's Abstract Machine: A Tutorial Reconstruction. The MIT Press, Cambridge, Massachusetts, London, England,1991
3. Bistarelli S., Montanari U. and Rossi F., Semiring-Based Constraint Satisfaction and Optimization. In *Journal of the ACM*, 44(2), pp:201-236, 1997.
4. Codognet P. and Diaz D., Compiling Constraints in *clp(FD)*. In *The Journal of Logic Programming*, 27, pp:185-226, 1996.
5. Davey B.A. and Priestley H.A., Introduction to Lattices and Order. Cambridge University Press, England, 1990.
6. Diaz D. and Codognet P., A minimal extension of the WAM for clp(FD). In Proc. of the *10th International Conference on Logic Programming (ICLP'93)*, pp:774-790,1993.
7. Frühwirth T., Theory and practice of constraint handling rules. In *The Journal of Logic Programming*, 37, pp:95-138, 1998.
8. Fernández A.J. and Hill P.M., A Design for a Generic Constraint Solver for Ordered Domains. In Proc. of *TCLP'98:Types for Constraint Logic Programming*, a JICSLP'98 Post Conference Workshop, Manchester, 1998.
9. Sidebottom G. and Havens, W.S., Hierarchical Arc Consistency for Disjoint Real Intervals in Constraint logic programming. In Computational Intelligence 8(4), 1992.

Proofs of Properties of the Operational Schema

(1) Termination. Let S_i and S_i' denote the constraint stores S and S', respectively, at the start of the of the $i+1$ iteration of the *while* loop. Then, S_0 is obtained by the initial stabilisation step for C and, for $i \geq 1$, S_i is obtained by the stabilisation step (4) in the $i-1$'st iteration. Also, $S_0' = \emptyset$ and, for $i \geq 1$, $S_i' = S_{i-1}$, by step (3). Since both S_i and S_i' are stable wrt V, for each variable $x \in V$, there are unique simple constraints $c_x^s \in S_i$ and $c_x'^s \in S_i'$. By the contractance property (1) of Theorem 2, $c_x^s \preceq c_x'^s$, for each $x \in V$. Thus, at the start of the $i+1$'st iterations of the while loop, because of the monotonicity condition (ii) for the $\mathit{precision}/1$ function, $\mathit{precision}(S_{i-1}) \geq \mathit{precision}(S_i)$. Thus using the extended version of step (1) that allows for a precision test, if there is an $i+1$ iteration, $\mathit{precision}(S_{i-1}) \geq \mathit{precision}(S_i) + \varepsilon$. Thus, $\mathit{precision}(S_0) \geq \mathit{precision}(S_i) + i \times \varepsilon$. However, for some $k \geq 0$, $\mathit{precision}(S_0) < k \times \varepsilon$, so that the procedure must terminate after no more than k iterations of the while loop.

(2) Correctness. Suppose the procedure terminates after k iterations. (If there are no iterations, then $C = \emptyset$ and the result is trivial.) We denote by S_i^s the set of all simple constraints in S_i, $0 \leq i \leq k$. Suppose S_i^s is propagated to C_i^s so that C_i^s is the set of simple constraints obtained in step (2) of the i'th iteration of the procedure. We need to show that if the procedure does not fail, then S_k^s is a solution for C. We show, by induction on i that S_i^s is stable wrt V and
(A) for $j \leq i$ and each $c_j^s \in S_j^s$, there exists $c_i^s \in S_i^s$ with the same constrained variable and $c_i^s \preceq_{R_D} c_j^s$.

When $i = 0$, then $C \mapsto S_0$ and, trivially $c_0^s \preceq_{R_D} c_0^s$. Suppose next that $i > 0$. Then $S_{i-1} \cup C_{i-1}^s \mapsto S_i^s$ so that S_i^s is stable wrt V. Moreover, by Definition 21 for each $c_{i-1}^s \in S_{i-1}^s$ there exists $c_i^s \in S_i^s$ with the same constrained variable and $c_i^s \preceq_{R_D} c_{i-1}^s$. However, by the induction hypothesis, if $j \leq i-1$ and $c_j^s \in C_j^s$ there is $c_{i-1}^s \in S_{i-1}^s$ with the same constrained variable and $c_{i-1}^s \preceq_{R_D} c_j^s$. Hence, for each $j \leq i$ and each $c_j^s \in C_j^s$, there exists $c_i^s \in S_i^s$ with the same constrained variable and $c_i^s \preceq_{R_D} c_j^s$. Letting $j = 0$ in (A), and using the fact that in the initialisation of the algorithm $C \mapsto S_0$, we obtain condition (1) in Definition 24.

We next prove that condition (2) in Definition 24 holds:
(B) for each $c^{ns} \in C^{ns}$ there is $c_k^s \in S_k^s$ with the same constrained variable and $c_k^s \preceq_{R_D} c^s$, where $c^{ns} \leadsto^{S_0} c^s$.

In the initialisation step for the algorithm, we have $C \mapsto S_0$. Then, in step (2), $S_0 \leadsto C_1^s$. Thus, by Definition 22, for each $c^{ns} \in C^{ns}$ (and hence also in S_0), there exists $c_1^s \in C_1^s$ and $c^{ns} \leadsto^{S_0} c_1^s$. Now, by step (4), $S_0 \cup C_1^s \mapsto S_1$ so that, by Definition 23, for each $c_1^s \in C_1^s$ there is $c^s \in S_1^s$ with the same constrained variable and $c_1^s \preceq_{R_D} c^s$. By (A) we have, for each $c_1^s \in S_1^s$ there is $c_k^s \in S_k^s$ with the same constrained variable and $c_k^s \preceq_{R_D} c_1^s$. Hence (B) holds.

By commutativity property of $\cap_D$ (see Subsection 3.6), for each $0 \leq i \leq k$, S_i is independent of the order in which the constraints in $S_{i-1} \cup C_{i-1} \mapsto S_i$ were intersected. Thus the solution S_k^s does not depend on the order in which the constraints were chosen.