An Interval Constraint System for Lattice Domains

ANTONIO J. FERNÁNDEZ
University of Málaga
and
PATRICIA M. HILL
University of Leeds

We present a generic framework for defining and solving interval constraints on any set of domains (finite or infinite) that are lattices. The approach is based on the use of a single form of constraint similar to that of an indexical used by CLP for finite domains and on a particular generic definition of an interval domain built from an arbitrary lattice. We provide the theoretical foundations for this framework and a schematic procedure for the operational semantics. Examples are provided that illustrate how new (compound) constraint solvers can be constructed from existing solvers using lattice combinators and how different solvers (possibly on distinct domains) can communicate and hence, cooperate in solving a problem. We describe the language $clp(\mathcal{L})$, which is a prototype implementation of this framework and discuss ways in which this implementation may be improved.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classification—*Constraint and logic languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Constraint, cooperation, indexicals, lattice, propagation

1. INTRODUCTION

Constraint Logic Programming (CLP) systems support many different domains such as finite ranges of integers [Carlsson et al. 1997; Codognet and Diaz 1996a], reals [Jaffar et al. 1992; Refalo and Van Hentenryck 1996; Sidebottom and Havens 1992; Benhamou 1995], finite sets [Walinsky 1989; Gervet 1997], or the Booleans [Codognet and Diaz 1996b; Barth and Bockmayr 1996]. The choice of

This work was supported by EPSRC grants GR/L19515 and GR/M05645; the first author was also supported by projects TIC2001-2705-C03-02 and TIC2002-04498-C05-02 funded by the Spanish Ministry of Science and Technology.

Authors' addresses: A. J. Fernández, Departamento de Lenguajes y Ciencias de la Computación, E.T.S.I.I., Universidad de Málaga, 29071 Teatinos, Málaga, Spain; email: afdez@lcc.uma.es; P. M. Hill, School of Computing, University of Leeds, Leeds, LS2 9JT, United Kingdom; email: hill@comp.leeds.ac.uk.

domain determines the nature of the constraints and their solvers; whether or not a domain is discrete or continuous as well as its cardinality influence the constraint solving procedures so that, for example, existing CLP systems have distinct constraint solving methods for the finite and the infinite domains. In practice, constraint problems are often not specific to any particular system domain and thus their formulation has to be artificially adapted to fit a given solver.

Most constraint solvers, called *black box* solvers, have the control fixed by the system. This black box approach enables very efficient implementations and can provide practical tools for the common constraint applications. However, such black box solvers lack adaptability for use in solving nonstandard problems. To overcome this lack of flexibility, many constraint systems provide *glass box* constraints [Frühwirth 1998; Codognet and Diaz 1996a]. These allow new constraints to be defined by the user. In this paper, we are particularly interested in glass box systems that do not require the user to have a detailed knowledge of the implementation.

From this perspective, there have been two main separate developments for the provision of glass box constraints: the constraint system clp(FD) [Codognet and Diaz 1996a] and the *Constraint Handling Rules (CHRs)* [Frühwirth 1998]. The first of these, designed for the finite domain (FD) of integers, is based on a single generic constraint that allows the user to define and control the constraint propagation. These constraints, often referred to as *indexicals*, are very efficient [Fernández and Hill 2000] since the implementation uses a simple interval narrowing technique which can be smoothly integrated into the WAM [Aït-kaci 1999; Diaz and Codognet 1993]. As a result, clp(FD) is now part of mainstream CLP systems such as SICStus Prolog [Carlsson et al. 1997], IF/Prolog [If/Prolog 1994], and GNU Prolog [Diaz and Codognet 2001]. On the other hand, the CHRs (now included as a library in SICStus Prolog [Carlsson et al. 1997]) enable the creation of new user-defined domains and their solvers and allow any interaction between them. Unfortunately the flexibility of these rules has an efficiency cost and the CHR systems have not been able to compete with other systems that employ the more traditional approaches [Fernández and Hill 2000].

It follows from this discussion that what is needed is a glass box system that combines the flexibility of CHRs with the efficiency of clp(FD). For this reason, we have adopted the indexical approach of clp(FD) to constraint solving, generalizing it for any set of domains that are lattices, thereby providing a flexibility closer to that of CHRs. Our framework has many advantages and, we believe, considerable potential as indicated below.

—The only condition we have placed on a domain is that it must be a lattice. Since, as far as we know, all existing domains provided for CLP systems are already lattices or could be easily extended to become lattices, it can support a wide variety of applications. Moreover, lattice combinators such as the direct and lexicographic products can be used to combine existing domains and their constraint operations to form the basis of new (compound) solvers.

—The framework is defined on a *set* of domains, allowing information to flow between domains. This provides an appropriate setting for solving (probably new) applications defined in the interval domain and over which different solvers, possibly on distinct domains, can communicate and, hence, cooperate.

—The basic schema for solving constraints is uniform over all domains regardless of whether they are user-defined or system-defined and irrespective of their cardinality. We prove that any constraint solver that is an instance of this schema is correct and show how termination may be ensured.

—Our framework is based on the indexical approach. Thus we anticipate that adaptations of the techniques used for the implementation of clp(FD) can be used for the implementation of our operational schema, thereby obtaining a reasonable efficiency compared with other CLP systems.

The paper is based on the Ph.D. dissertation of A. J. Fernández [Fernández 2002]; more information about the theory and/or the $clp(\mathcal{L})$ language/system is available there. Furthermore, the constraint framework described in this paper is a simplified and improved version of that given in Fernández and Hill [1999b]. In Section 2, we describe the algebraic concepts used in the paper. In Section 3, we define the computation domain and construct the interval domain used for constraint solving. In Section 4, the interval constraints are presented together with the procedure for constraint propagation and narrowing. In Section 5, we provide a schema for the operational semantics of our constraint solver and show how this can be adapted so as to ensure termination for infinite as well as finite domains. In Section 6, we provide additional nonstandard examples of computation domains and also show how new domains can be constructed using different lattice combinators. We also define high-level constraints and show how these provide support for solver cooperation. An overview of the CLP language $clp(\mathcal{L})$ that we have implemented to demonstrate the viability of the theoretical framework is described in Section 7. Section 8 deals with several important issues as for example how to solve disjunctive constraints or nonlinear constraints among others. The main part of the paper ends with a discussion about related work and the conclusions. The proofs of all the results stated in the main part of the paper are included in the Appendix.

2. PRELIMINARIES AND NOTATION

If C is a set, then $\#C$ denotes its cardinality, $\wp(C)$ its power set, and $\wp_f(C)$ the set of all the finite subsets of C, that is to say, $\wp_f(C) = \{c \in \wp(C) \mid c \text{ is finite}\}$.

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. 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. Also $c \prec c'$ if $c \preceq c'$ and $c \neq c'$. Any set C for which an ordering relation is defined is said to be *ordered*. We say C is *totally ordered* if for any $a, b \in C$, $a \sim b$.

Bounds. Let C be an ordered set. An element c in C is a *lower* (*upper*) *bound* of a subset $E \subseteq C$ if and only if $\forall x \in E : c \preceq x$ ($x \preceq c$). 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 $\mathrm{glb}_C(E)$ ($\mathrm{lub}_C(E)$). If $E = \{x, y\}$, we write $\mathrm{glb}_C(x, y)$ to denote $\mathrm{glb}_C(\{x, y\})$ and $\mathrm{lub}_C(x, y)$ to denote $\mathrm{lub}_C(\{x, y\})$.

Predecessor and successor. Let C be an ordered set and let $c, c' \in C$. Then $c = \mathrm{pre}(c')$ is an *immediate predecessor* of c' and $c' = \mathrm{succ}(c)$ an *immediate successor* of c if $c \prec c'$ and for any $c'' \in C$ with $c \preceq c'' \prec c'$ implies $c = c''$.

Monotonicity. Let f be a n-ary function $f :: C_1 \times \cdots \times C_n \to C$, where C and all C_i, for $i \in \{1, \ldots, n\}$, are ordered sets. Then f is *monotonic in* C if, whenever $t_i, t'_i \in C_i$ such that $t_i \preceq t'_i$, for all $i \in \{1, \ldots, n\}$, then

$$f(t_1, \ldots, t_i, \ldots, t_n) \preceq f(t'_1, \ldots, t'_i, \ldots, t'_n).$$

A monotonic function f is *strict monotonic* if, whenever $t_i, t'_i \in C_i$ such that $t_i \preceq t'_i$, for all $i \in \{1, \ldots, n\}$ and $t_j \prec t'_j$, for some $j \in \{1, \ldots, n\}$, then

$$f(t_1, \ldots, t_i, \ldots, t_n) \prec f(t'_1, \ldots, t'_i, \ldots, t'_n).$$

Lattice. Let L be an ordered set. L is a *lattice* if $\mathrm{lub}_L(x, y)$ and $\mathrm{glb}_L(x, y)$ exist, for any two elements $x, y \in L$.

Top and bottom elements. Let L be a lattice. If it exists, $\mathrm{glb}_L(L) = \bot_L$ is *the bottom element* of L. Similarly, if it exists, $\mathrm{lub}_L(L) = \top_L$ is *the top element* of L. The lack of a bottom or top element can be remedied by adding a fictitious one. Thus, the *lifted lattice* of L is $L \cup \{\bot_L, \top_L\}$ where, if $\mathrm{glb}_L(L)$ 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 $\mathrm{lub}_L(L)$ does not exist, $\top_L$ is a new element not in L such that $\forall a \in L, a \prec \top_L$.

Dual. Let L be a lattice. If $a \in L$, then we denote its dual as $\hat{a}$. The *dual of* L, denoted by $\hat{L}$, is the lattice that contains the dual element of any element in L, that is to say, $\hat{L} = \{\hat{a} \mid a \in L\}$, and where the ordering is reversed with respect L, that is to say, if $a, b \in L$, then $\hat{a} \preceq \hat{b}$ if and only if $b \preceq a$. As consequence, $\hat{L}$ is the lattice that contains exactly the same elements as L and that is obtained by interchanging $\mathrm{glb}_L(a, b)$ and $\mathrm{lub}_L(a, b)$ for any $a, b \in L$. The *duality principle for lattices* is "the dual of a statement about lattices phrased in terms of glb and lub can be obtained simply by interchanging glb and lub."

Products. 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 where

$$\begin{aligned}
\mathrm{glb}(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) &= \langle \mathrm{glb}_{L_1}(x_1, y_1), \mathrm{glb}_{L_2}(x_2, y_2)\rangle;\\
\mathrm{glb}((x_1, x_2), (y_1, y_2)) &= \text{if } x_1 = y_1 \text{ then } (x_1, \mathrm{glb}_{L_2}(x_2, y_2))\\
&\qquad \text{elsif } x_1 \prec y_1 \text{ then } (x_1, x_2)\\
&\qquad \text{elsif } x_1 \succ y_1 \text{ then } (y_1, y_2)\\
&\qquad \text{else } (\mathrm{glb}_{L_1}(x_1, y_1), \top_{L_2});
\end{aligned}$$

lub is the dual of glb;

$$\top_{\langle L_1,L_2\rangle} = \langle \top_{L_1}, \top_{L_2}\rangle \quad \text{and} \quad \bot_{\langle L_1,L_2\rangle} = \langle \bot_{L_1}, \bot_{L_2}\rangle;$$
$$\top_{(L_1,L_2)} = (\top_{L_1}, \top_{L_2}) \quad \text{and} \quad \bot_{(L_1,L_2)} = (\bot_{L_1}, \bot_{L_2}).$$

Moreover,

$$\langle x_1, y_1\rangle \preceq \langle y_1, y_2\rangle \quad \text{iff} \quad x_1 \preceq y_1 \text{ and } x_2 \preceq y_2;$$
$$(x_1, y_1) \preceq (x_2, y_2) \quad \text{iff} \quad x_1 \prec x_2 \text{ or } x_1 = x_2 \text{ and } y_1 \preceq y_2.$$

For more information about lattices see, for example, Davey and Priestley [1990]. In the rest of the paper, $(L, \preceq, \text{glb}_L, \text{lub}_L, \bot_L, \top_L)$ denotes a (possible lifted) lattice on L with (possibly fictitious) bounds $\bot_L$ and $\top_L$.

3. THE COMPUTATION AND INTERVAL DOMAINS

The domain on which the values are actually computed is called a *computation domain*. The key aspect of the constraint system described in this paper is that it can be built on any computation domain provided it is a lattice. Throughout the paper, we let $\mathcal{L}$ denote a (possibly infinite) set of computation domains containing at least one element L and let $\hat{\mathcal{L}} = \{\hat{L} \mid L \in \mathcal{L}\}$. With each computation domain $L \in \mathcal{L}$, we associate a set of variable symbols V_L that is disjoint from $V_{L'}$ for any $L' \in \mathcal{L}$ distinct of L. We define $\mathcal{V}_{\mathcal{L}} = \cup\{V_L | L \in \mathcal{L}\}$. It is assumed (without loss of generality) that all $L \in \mathcal{L}$ are lifted lattices.

Example 1. Most classical constraint domains are lattices. For instance,

$$(\mathit{Integer}, \leq, \text{mini}, \text{maxi}, \bot_{\mathit{Integer}}, \top_{\mathit{Integer}}),$$
$$(\Re, \leq, \text{mini}, \text{maxi}, \bot_{\Re}, \top_{\Re}),$$
$$(\mathit{Bool}, \Rightarrow, \wedge, \vee, \mathit{false}, \mathit{true}),$$
$$(\mathit{Set}\ L, \subseteq, \cap, \cup, \emptyset, L)$$

are lattices for the integers, reals, Booleans, and sets, respectively, under their usual orders where mini and maxi functions return, respectively, the minimum and maximum element of any two elements in the integers or reals. Note that *Integer* and $\Re$ are lifted lattices and include the fictitious elements $\top_{\mathit{Integer}}$, $\bot_{\mathit{Integer}}$, $\top_{\Re}$, and $\bot_{\Re}$. For the Booleans, it is assumed that $\mathit{Bool} = \{\mathit{false}, \mathit{true}\}$. For the set lattice, we assume that $\mathit{Set}\ \ L = \wp(L)$, for each $L \in \mathcal{L}$, where $\mathcal{L} = \{\mathit{Integer}, \Re, \mathit{Bool}\} \cup \{\mathit{Set}\ \ L \mid L \in \mathcal{L}\}$. Note that $\mathcal{L}$ is an infinite set of computations domains.

In the rest of the examples in the paper, we will use the computation domains *Integer*, $\Re$, *Bool* and *Set L* for some domain $L \in \mathcal{L}$ without further comment or direct reference to this example.

Although the framework for our constraint system is based on the indexical approach of clp(FD) to constraint solving which propagates constraints on finite closed intervals, this framework is intended for all lattices including infinite and continuous ones. For this reason, we need to be able to define the constraints over both open and closed intervals. Thus, in the rest of this section, we will show how a computation domain can be combined with a special binary lattice we call the *bracket domain* to form an *interval domain* suitable for the constraint solving mechanism described in Section 4. We do this in a number of stages. First, in Section 3.1, we define the bracket domain and use this to construct *bounded computational domains* for the left and right bounds of an interval. In Section 3.2, these are extended to allow for constraint operators. These are further extended in Section 3.3 to include an additional construct called an *indexical*. In Section 3.4, we use this enhanced bounded domain to construct the *interval domain*.

3.1 Bounded Computation Domains

To define open and closed bounds of the intervals, we first define a domain of brackets.

Definition 1 (*Bracket Domain*). The *bracket domain* B is the lattice

$$(\{`)\text{'}, `]\text{'}\}, \prec, \min_B, \max_B, `)\text{'}, `]\text{'}),$$

where the ordering $\prec$ is defined by ‘)’ $\prec$ ‘]’, and $\min_B$ and $\max_B$ functions return, respectively, the minimum and maximum of any two elements in B.

This domain is combined with any computation domain to form the right and left bounds of an interval.

Definition 2 (*Simple Bounded Computation Domain*). The *simple bounded computation domain* for L is the lattice resulting from the lexicographic product (L, B) and denoted by L^s. An element $(a, b) \in L^s$ is denoted by a_b.

The *mirror* of L^s is the lexicographic product $(\hat{L}, B)$ and is denoted by $\overline{L^s}$. The *mirror* of an element $t = (a, b) \in L^s$ is $(\hat{a}, b) \in \overline{L^s}$ and denoted by $\overline{t} = \overline{a_b}$.

The simple bounded computation domain and its mirror maintain some useful relations that are independent of any specific computation domain. For instance, it follows directly from the definition that $\overline{L^s} = \widehat{L^s}$. Moreover, if $t_1 = a_b, t_2 = c_d \in L^s$, then

$$\text{if } a \neq c, \left\{ \begin{array}{c} \overline{t_2} \prec \overline{t_1} \iff t_1 \prec t_2 \\ \text{glb}_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{\text{lub}_{L^s}(t_1, t_2)} \\ \text{lub}_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{\text{glb}_{L^s}(t_1, t_2)} \end{array} \right\} \text{ else } \left\{ \begin{array}{c} \overline{t_1} \preceq \overline{t_2} \iff t_1 \preceq t_2 \\ \text{glb}_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{\text{glb}_{L^s}(t_1, t_2)} \\ \text{lub}_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{\text{lub}_{L^s}(t_1, t_2)} \end{array} \right\}.$$

Example 2. When $L = Integer$, $6_]$ denotes (6, ‘]’) and $\overline{6_]}$ denotes ($\hat{6}$, ‘]’). Also

$$\begin{array}{l} 0_) \prec 0_] \prec 1_) \prec 1_] \prec \cdots \prec \top_{L_)} \prec \top_{L_]} \text{ in } Integer^s, \\ \overline{\top_{L_)}} \prec \overline{\top_{L_]}} \prec \cdots \prec \overline{1_)} \prec \overline{1_]} \prec \overline{0_)} \prec \overline{0_]} \text{ in } \overline{Integer^s}, \\ \text{glb}_{L^s}(3_], 5_]) = \text{lub}_{L^s}(3_], 3_)) = 3_], \\ \text{glb}_{\overline{L^s}}(\overline{3_]}, \overline{5_]}) = \overline{\text{lub}_{L^s}(3_], 5_])} = \overline{5_]} = (\hat{5}, `]\text{'}), \\ \text{lub}_{\overline{L^s}}(\overline{3_]}, \overline{5_]}) = \overline{\text{glb}_{L^s}(3_], 5_])} = \overline{3_]} = (\hat{3}, `]\text{'}). \end{array}$$

Moreover, when $L = \mathit{Set\ Integer}$, $\{1, 3\}_)$ denotes $(\{1, 3\}, \text{‘)’})$ and $\overline{\{1, 3\}_)}$ denotes $(\widehat{\{1, 3\}}, \text{‘)’})$. Also

$$\begin{aligned}
&\{1\}_] \prec \{1, 3\}_) \prec \{1, 3\}_] \prec \{1, 3, 5\}_] \text{ in } \mathit{Set\ Integer}^s,\\
&\overline{\{1, 3, 5\}_]} \prec \overline{\{1, 3\}_)} \prec \overline{\{1, 3\}_]} \prec \overline{\{1\}_]} \text{ in } \overline{\mathit{Set\ Integer}^s},\\
&\mathrm{glb}_{L^s}(\{4\}_], \{4, 6\}_]) = \{4\}_] \text{ and } \mathrm{lub}_{L^s}(\{3\}_], \{4, 6\}_]) = \{3, 4, 6\}_],\\
&\mathrm{lub}_{\overline{L^s}}(\overline{\{4\}_]}, \overline{\{4, 6\}_)}) = \overline{\mathrm{glb}_{L^s}(\{4\}_], \{4, 6\}])} = \overline{\{4\}_]} = (\{\hat{4}\}, \text{‘]’}).
\end{aligned}$$

3.2 Constraint Operators

The bounded computation domains are extended to allow for operators.

Definition 3 (*Constraint Operators*). Suppose $L \in \mathcal{L}$ and $L_1, \ldots, L_n \in \mathcal{L} \cup \hat{\mathcal{L}}$. Then $\circ$ is called a *constraint operator for* L^s if it is defined as

$$\begin{gathered}
\circ :: L_1^s \times \cdots \times L_n^s \to L^s,\\
\circ(a_{1b_1}, \ldots, a_{nb_n}) = \circ_L(a_1, \ldots, a_n)_{\circ_B(b_1,\ldots,b_n)},
\end{gathered} \tag{1}$$

where $a_i \in L_i$ and $b_i \in B$, for $1 \le i \le n$, and $\circ_L$ and $\circ_B$ are monotonic functions

$$\begin{gathered}
\circ_L :: L_1 \times \cdots \times L_n \to L,\\
\circ_B :: \underbrace{B \times \cdots \times B}_{n \text{ times}} \to B,
\end{gathered}$$

defined in the computation domain L and in the bracket domain, respectively, in such a way that, if $\circ_L$ is not a strict monotonic function then $\circ_B$ is a constant function. The *mirror* $\overline{\circ}$ of $\circ$ is then defined as

$$\begin{gathered}
\overline{\circ} :: \overline{L_1^s} \times \cdots \times \overline{L_n^s} \to \overline{L^s},\\
\overline{\circ}(\overline{t_1}, \ldots, \overline{t_i}, \ldots, \overline{t_n}) = \overline{\circ(t_1, \ldots, t_i, \ldots, t_n)}.
\end{gathered} \tag{2}$$

If $\circ$ is a binary constraint operator, we often use infix notation and write $t_1 \circ t_2$ instead of $\circ(t_1, t_2)$.

Example 3. Suppose for any $b_1, b_2 \in B$,

$$b_1 +_B b_2 = \min\nolimits_{\mathrm{B}}(b_1, b_2), \qquad b_1 -_B b_2 = \text{‘]’ if } b_1 = b_2 \text{ and ‘)’ otherwise.}$$

Suppose also that $+_L :: L \times L \to L$ and $-_L :: L \times \hat{L} \to L$ are strict monotonic on L. Then the binary constraint operators $+$, $-$, and their mirrors are defined as

$$\begin{aligned}
+ &:: L^s \times L^s \to L^s, & - &:: L^s \times \overline{L^s} \to L^s,\\
a_b + c_d &= (a +_L c)_{b +_B d}, & a_b - \overline{c_d} &= (a -_L \hat{c})_{b -_B d},\\[1em]
\overline{+} &:: \overline{L^s} \times \overline{L^s} \to \overline{L^s}, & \overline{-} &:: \overline{L^s} \times L^s \to \overline{L^s},\\
\overline{a_b}\,\overline{+}\,\overline{c_d} &= \overline{(a +_L c)_{b +_B d}}, & \overline{a_b}\,\overline{-}\,c_d &= \overline{(a -_L \hat{c})_{b -_B d}}.
\end{aligned}$$

Consider the case $L \in \{Integer, \Re\}$; then we define $+_L/2$ and $-_L/2$ to return, respectively, the sum and difference of their arguments. For example, when $L = \Re$, by Equations (1) and (2), we have

$$3.2_{)} + 4.1_{]} \;=\; (3.2 +_{\Re} 4.1)_{)+_B]} \;=\; 7.3_{)}, \qquad \overline{3.2_{)}}\,\overline{+}\,\overline{4.1_{]}} \;=\; \overline{3.2_{)} + 4.1_{]}} \;=\; \overline{7.3_{)}}.$$
$$7.3_{)} - \overline{4.1_{]}} \;=\; (7.3 - \widehat{4.1})_{)-_B]} \;=\; 3.2_{)}, \qquad \overline{7.3_{)}}\,\overline{-}\,4.1_{]} \;=\; \overline{7.3_{)} - \overline{4.1_{]}}} \;=\; \overline{3.2_{)}}.$$

Observe that we declared $-_L$ with the second argument in the mirror domain of that of the first. This ensures that operators such as $-_{Integer}$, $-_{\Re}$ are monotonic on both arguments. For instance, if $i \in Integer$, then, as $\hat{\imath}$ increases, $10 - \hat{\imath}$ increases but $10 - i$ decreases. From the operator declarations our framework will ensure that all constraints are propagated monotonically (see Section 4.3).

Suppose that the unary operator trunc is defined as

$$\text{trunc} :: \Re^s \to Integer^s,$$
$$\text{trunc}(a_b) = \text{trunc}_{Integer}(a)_{\text{trunc}_B(b)},$$

where $\text{trunc}_{Integer}(a)$ is defined to return the integer part of a, for any a $\in \Re$, and $\text{trunc}_B(b) =$ ']' for any $b \in B$. Then

$$\overline{\text{trunc}} :: \overline{\Re^s} \to \overline{Integer^s}.$$

Thus, by Equations (1) and (2),

$$\text{trunc}(3.1_{)}) \;=\; \text{trunc}_{Integer}(3.1)_{\text{trunc}_B(')')} \;=\; 3_{]},$$
$$\overline{\text{trunc}}(\overline{3.1_{)}}) \;=\; \overline{\text{trunc}(3.1_{)})} \;=\; \overline{3_{]}}.$$

Consider now the case $L = Set\ L'$ with $L' \in \mathcal{L}$; then we define $+/2$ and $-/2$ as follows:

$$+ :: L^s \times L^s \to L^s, \qquad - :: L^s \times \overline{L^s} \to L^s,$$
$$a_b + c_d = (a \cup c)_{]}, \qquad a_b - \overline{c_d} = (a \backslash \hat{c})_{]}.$$

Then, for example, for $L' = Integer$,

$$\{1, 3\}_{]} + \{3, 4\}_{)} \;=\; (\{1, 3\} \cup \{3, 4\})_{]} \;=\; \{1, 3, 4\}_{]},$$
$$\overline{\{1, 3\}_{]}}\,\overline{+}\,\overline{\{3, 4\}_{)}} \;=\; \overline{\{1, 3\}_{]} \cup \{3, 4\}_{)}} \;=\; \overline{\{1, 3, 4\}_{]}},$$
$$\{1, 3\}_{]} - \overline{\{3, 4\}_{]}} \;=\; (\{1, 3\} \backslash \widehat{\{3, 4\}})_{]} \;=\; \{1\}_{]},$$
$$\overline{\{1, 3\}_{]}}\,\overline{-}\,\{3, 4\}_{]} \;=\; \overline{\{1, 3\}_{]} \backslash \overline{\{3, 4\}_{]}}} \;=\; \overline{\{1\}_{]}}.$$

In the definition of an operator such as trunc in the previous example, we allowed for its arguments to have a different computation domain from that of the result. This provides a channel of communication from one solver to another, allowing the information to flow between different domains.

3.3 Indexicals

The variables in V_L are introduced into the domain L^s and its mirror by means of *indexicals*. Let O_L be a set of constraint operators defined for L^s.

Definition 4 (*Bounded Computation Domain*). The *bounded computation domain* (*for* L) L^b and its *mirror* $\overline{L^b}$ are defined as

$$
\begin{aligned}
L^b &= L^s \cup \{\max(x) \mid x \in V_L\} \cup \{\mathrm{val}(x) \mid x \in V_L\} \\
&\quad \cup \left\{ \circ(t_1, \dots, t_n) \;\middle|\; \begin{array}{l} \circ :: L'_1 \times \cdots \times L'_n \to L^s \in O_L, \\ t_i \in (L'_i)^b \text{ for } (1 \le i \le n) \end{array} \right\}, \\
\overline{L^b} &= \{\overline{t} \mid t \in L^b\},
\end{aligned}
$$

where

$$
\begin{array}{l}
\text{for} \quad i \in \{1, \dots, n\}, (L'_i)^b = L^b_i \quad \text{if} \quad L'_i = L^s_i \text{ and } (L'_i)^b = \overline{L^b_i} \text{ if } L'_i = \overline{L^s_i}; \\
\overline{\max(x)} = \min(x),\ \overline{\min(x)} = \max(x) \text{ and } \overline{\overline{\mathrm{val}(x)}} = \mathrm{val}(x); \\
\text{for each} \quad \circ :: L'_1 \times \cdots \times L'_n \to L^s \in O_L,
\end{array}
$$

$$\overline{\circ(t_1, \dots, t_i, \dots, t_n)} = \overline{\circ}(\overline{t_1}, \dots, \overline{t_i}, \dots, \overline{t_n}).$$

The expressions $\max(x)$, $\min(x)$, $\mathrm{val}(x)$, and $\overline{\mathrm{val}(x)}$ are called *indexicals*.

We define $\top_{L^b} = \top_{L^s}$ and $\bot_{L^b} = \bot_{L^s}$ and the ordering of L_b to be inherited from that of L^s. Thus L^b is also a lattice. Note that, if $t \in L^b$, then $\overline{t} \in \overline{L^b}$, and, if $t \in \overline{L^b}$, then $\overline{t} \in L^b$.

Example 4. Let $+$ (for $L = \mathit{Integer}$), $-$ (for $L = \Re$), and trunc be as defined in Example 3. Then $+$, $\mathrm{trunc} \in O_{\mathit{Integer}}$, $- \in O_{\Re}$. Let also $i \in V_{\mathit{Integer}}$ and $r \in V_{\Re}$; then

$$
\begin{array}{rllll}
3_], & \max(i), & 3_) + \max(i), & \mathrm{trunc}(\max(r)) & \text{are in } \mathit{Integer}^b; \\
20.1_), & \max(r), & 20.1_) - \min(r), & \mathrm{val}(r) & \text{are in } \Re^b; \\
\overline{3_]}, & \min(i), & \overline{3_)} \mathbin{\overline{+}} \min(i), & \overline{\mathrm{trunc}}(\min(r)) & \text{are in } \overline{\mathit{Integer}^b}; \\
\overline{20.1_)}, & \min(r), & \overline{20.1_)} \mathbin{\overline{-}} \max(r), & \overline{\mathrm{val}(r)} & \text{are in } \overline{\Re^b};
\end{array}
$$

and

$$\overline{3_) + \max(i)} = \overline{3_)} \mathbin{\overline{+}} \overline{\max(i)} = \overline{3_)} \mathbin{\overline{+}} \min(i).$$

3.4 Interval Domains

We now define the structure over which the constraints will be solved.

Definition 5 (*Interval Domain*). The *interval domain* R^b_L over the domain L is the direct product $\langle \overline{L^b}, L^b \rangle$. The *simple interval domain* R^s_L over L is the direct product $\langle \overline{L^s}, L^s \rangle$.

Note that R^b_L and R^s_L are lattices since they are constructed from the direct product of lattices. It is important to note that the ordering $\preceq$ for R^s_L simulates the interval inclusion.[1]

[1]For instance, $\langle \overline{3.0_]}, 4.0_) \rangle \preceq \langle \overline{1.8_)}, 4.5_] \rangle$ intuitively means that $[3.0, 4.0) \subseteq (1.8, 4.5]$.

Definition 6 (*Range*). An element $r \in R^b_L$ is called a *range*. If $r \in R^s_L$, then we say it is *simple*. A simple range $r = \langle \overline{s}, t\rangle$ is *consistent*

(1) if $s \preceq t$ and

(2) if $s = a_)$ and, for some $b \in B$, $t = a'_b$, then $a \neq a'$.

Note that (1) implies that r is inconsistent if $s \not\preceq_{L^s} t$.

Example 5. In the domains $\Re$, *Integer*, and *Set Integer*, with $i \in V_\Re$ and $s \in V_{Set\ Integer}$, and where $+$ (for $L \in \{\Re, Set\ Integer\}$) is as defined in Example 3,

$$\begin{aligned}
\langle \overline{2.3_]}, 8.9_)\rangle &\in R^s_\Re \text{ is consistent;}\\
\langle \overline{2.3_]}, 2.2_]\rangle &\in R^s_\Re \text{ is inconsistent;}\\
\langle \overline{1.4_]}, \max(i) + 4.9_]\rangle &\in R^b_\Re;\\
\mathrm{glb}_{R^b_\Re}(\langle \overline{3.2_]}, 6.7_]\rangle, \langle \overline{1.8_)}, 4.5_]\rangle) = \langle \overline{3.2_]}, 4.5_]\rangle&;\\
\mathrm{lub}_{R^b_\Re}(\langle \overline{3.2_]}, 6.7_]\rangle, \langle \overline{1.8_)}, 4.5_]\rangle) = \langle \overline{1.8_)}, 6.7_]\rangle&;\\
\langle \overline{1_]}, 10_]\rangle &\in R^s_{Integer} \text{ is consistent;}\\
\langle \overline{1_)}, 1_]\rangle, \langle \overline{5_]}, 2_]\rangle &\in R^s_{Integer} \text{ are inconsistent;}\\
\langle \overline{\{1\}_]}, \{1, 3\}_]\rangle &\in R^s_{Set\ Integer} \text{ is consistent;}\\
\langle \overline{\{1, 3\}_]}, \{1\}_]\rangle, \langle \overline{\{1, 3\}_]}, \{1, 4\}_]\rangle &\in R^s_{Set\ Integer} \text{ are inconsistent;}\\
\langle \overline{\{1\}_]} \overline{+} \min(s), \{2, 5, 7, 9\}_]\rangle &\in R^b_{Set\ Integer}.
\end{aligned}$$

The next result shows that given an (in)consistent range in the R^s_L lattice it is possible to identify a part of this lattice where every range is (in)consistent.

PROPOSITION 1. *Suppose $r, r' \in R^s_L$, for any $L \in \mathcal{L}$, where $r \preceq r'$. If r' is inconsistent, then r is also inconsistent.*

Example 6. Suppose $L \in \mathcal{L}$ and $a, b, c \in L$ where $a \prec c \prec b$. Figure 1 illustrates the part of lattice R^s_L constructed from the elements a, b and c. Note that the nodes within the square are all inconsistent ranges where the rest of nodes are all consistent. The nodes with circles are special cases and are considered in Section 8.1.

4. THE CONSTRAINT DOMAINS

Interval constraints, which are the basic elements of the constraint solver, are defined in Section 4.1 by coupling a variable with a range. In Section 4.2, we define constraint stores and show how a solution can be computed using two procedures: constraint stabilization and constraint propagation. Then, in Section 4.3, we explain how our constraint system enforces the monotonic propagation of constraints.

As in previous sections, L denotes any domain in $\mathcal{L}$, V_L the set of variables associated with L, $\mathcal{V}_\mathcal{L} = \cup\{V_L | L \in \mathcal{L}\}$, and R^b_L the interval domain over L. Let $X \in \wp_f(\mathcal{V}_\mathcal{L})$.

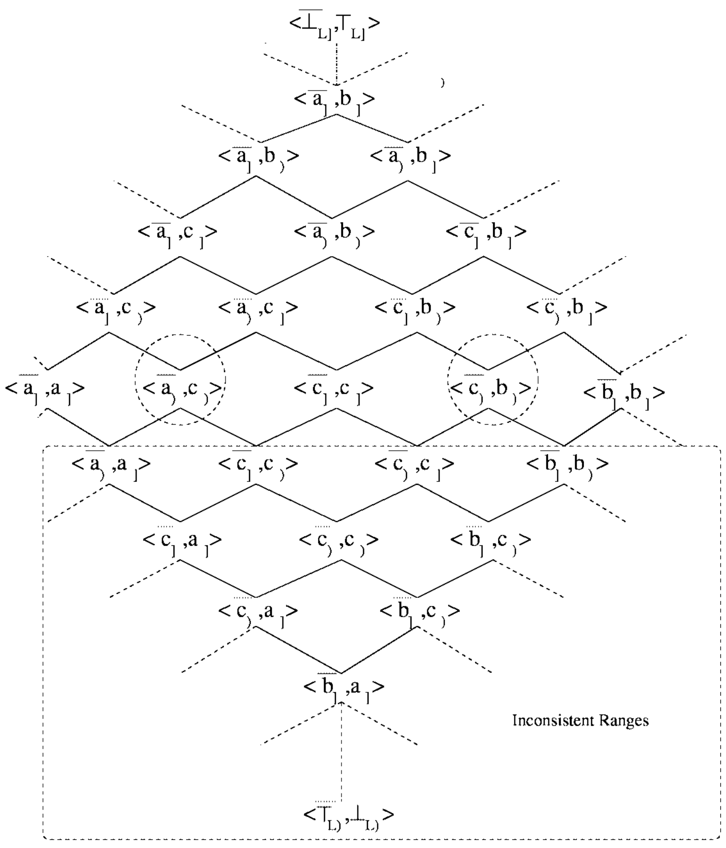

Fig. 1. Structure of the simple interval domain R^s_L where $a, b, c \in L$ and $a \prec c \prec b$.

4.1 Interval Constraints

Definition 7 (*Interval Constraint Domain*). Suppose $\sqsubseteq :: V_L \times R^b_L$. Then, for all $x \in V_L$ and $r \in R^b_L$,

$$c = x \sqsubseteq r$$

is called an *interval constraint* for L with *constrained variable* x. If r is simple (respectively consistent), then c is *simple* (respectively *consistent*). If $r = \top_{R^b_L}$, then c is called a *type constraint for* x and denoted by $x ::' L$. If $t \in L$, then $x = t$ is a shorthand for $x \sqsubseteq \langle \overline{t_]}, t_] \rangle$. The *interval constraints domain over* X *for* L is the set of all interval constraints for L with constrained variables in X and is denoted by $\mathcal{C}^X_L$. The union

$$\mathcal{C}^X \stackrel{\text{def}}{=} \bigcup \{\mathcal{C}^X_L \mid L \in \mathcal{L}\}$$

is called the *interval constraint domain over* X for $\mathcal{L}$.

The ordering for $\mathcal{C}^X$ is inherited from the ordering in R^b_L. We define $c_1 \preceq c_2$ if and only if, for some $L \in \mathcal{L}$, $c_1 = x \sqsubseteq r_1, c_2 = x \sqsubseteq r_2 \in \mathcal{C}^X_L$ and $r_1 \preceq r_2$.

Definition 8 (*Intersection of Simple Interval Constraints*). Suppose $x \in X$. The intersection in a domain $L \in \mathcal{L}$ of two simple constraints $c_1, c_2 \in \mathcal{C}_L^X$ where $c_1 = x \sqsubseteq r_1$, $c_2 = x \sqsubseteq r_2$, and $x \in V_L$ is defined as follows:

$$c_1 \cap_L c_2 = \text{glb}_{\mathcal{C}_L^X}(c_1, c_2) = x \sqsubseteq \text{glb}_{R_L^s}(r_1, r_2).$$

Suppose $x \in X$ and $c_1, c_2, c_3 \in \mathcal{C}_L^X$ are simple interval constraints with constrained variable x and $c_3 = c_1 \cap_L c_2$. Then it follows from the definition that $\cap_L$ has the following properties:

Contractance: $c_3 \preceq c_1$ and $c_3 \preceq c_2$;
Correctness: if $c \preceq c_1$ and $c \preceq c_2$, then $c \preceq c_3$;
Commutativity: $(c_1 \cap_L c_2) = (c_2 \cap_L c_1)$;
Idempotence: $(c_1 \cap_L c_3) = c_3$ and $(c_3 \cap_L c_2) = c_3$.

If $S_x \subseteq \mathcal{C}_L^X$ is a set of simple constraints with constrained variable x, then we define $\bigcap_L S_x = \text{glb}_{\mathcal{C}_L^X}(S_x)$. As a result of the contractance property, we have that $\bigcap_L S_x \preceq c$, for each $c \in S_x$.

Example 7. Examples of the intersection of simple interval constraints are

$$\begin{aligned} i \sqsubseteq \langle\overline{5_]}, 24_]\rangle \cap_{Integer} i \sqsubseteq \langle\overline{1_]}, 14_]\rangle &= \text{glb}_{\mathcal{C}^X_{Integer}}(i \sqsubseteq \langle\overline{5_]}, 24_]\rangle, i \sqsubseteq \langle\overline{1_]}, 14_]\rangle) \\ &= i \sqsubseteq \text{glb}_{R^s_{Integer}}(\langle\overline{5_]}, 24_]\rangle, \langle\overline{1_]}, 14_]\rangle) \\ &= i \sqsubseteq \langle \text{glb}_{\overline{Integer^s}}(\overline{5_]}, \overline{1_]}), \text{glb}_{Integer^s}(24_], 14_])\rangle \\ &= i \sqsubseteq \langle\overline{5_]}, 14_]\rangle. \end{aligned}$$

$$\begin{aligned} r \sqsubseteq \langle\overline{1.12_]}, 5.67_)\rangle \cap_{\Re} r \sqsubseteq \langle\overline{2.34_]}, 5.95_)\rangle &= r \sqsubseteq \langle\overline{2.34_]}, 5.67_)\rangle; \\ b \sqsubseteq \langle\overline{false_)}, true_]\rangle \cap_{Bool} b \sqsubseteq \langle\overline{false_]}, true_]\rangle &= b \sqsubseteq \langle\overline{false_)}, true_]\rangle; \\ s \sqsubseteq \langle\overline{\{1\}_]}, \{1,2,3\}_]\rangle \cap_{Set\ Integer} s \sqsubseteq \langle\overline{\{2\}_]}, \{1,2,4\}_]\rangle &= s \sqsubseteq \langle\overline{\{1,2\}_]}, \{1,2\}_]\rangle. \end{aligned}$$

4.2 Constraint Stores

Definition 9 (*Constraint Store*). If $S \in \wp_f(\mathcal{C}^X)$, then S is a *constraint store* for X. If S contains only simple constraints, then it is *simple*. If S is simple, then it is *consistent* if all its constraints are consistent. The set of all simple constraint stores for X is denoted by $\mathcal{S}^X$. A constraint store S is *stable* if there is exactly one simple constraint for each $x \in X$ in S. The set of all simple stable constraint stores for X is denoted by $\mathcal{SS}^X$.

Let $S, S' \in \mathcal{SS}^X$ where c_x, c'_x denote the (simple) constraints for $x \in X$ in S and S', respectively. Then $S \preceq S'$ if and only if, for each $x \in X$, $c_x \preceq c'_x$. Let $\top_{\mathcal{SS}^X}$ be the set of type constraints for X and $\bot_{\mathcal{SS}^X} = \{x \sqsubseteq \bot_{R_L^s} \mid x \in X \cap V_L, L \in \mathcal{L}\}$. Then, with these definitions, $\mathcal{SS}^X$ forms a lattice.

A solution to a constraint store is obtained as a combination of two processes: *constraint propagation* and *constraint stabilization* (i.e., constraint narrowing). The first of this, constraint propagation, is defined by means of an evaluation function.

Definition 10 (*Evaluating Interval Constraints*). Let $S \in \mathcal{SS}^X$, $x \in X$, and let

$$\begin{aligned}\mathcal{L}^X &= \cup\{L^X \mid L \in \mathcal{L}, L^X = \{t \in L^b \mid \text{vars}(t) \subseteq X\}\},\\ \overline{\mathcal{L}^X} &= \cup\{\overline{L^X} \mid L \in \mathcal{L}, \overline{L^X} = \{t \in \overline{L^b} \mid \text{vars}(t) \subseteq X\}\},\end{aligned}$$

where vars(t) denotes the set of variables occurring in t. Then the (overloaded) evaluation functions are defined:

$$\text{eval} :: \mathcal{SS}^X \times \mathcal{L}^X \to \mathcal{L}^X, \qquad \text{eval} :: \mathcal{SS}^X \times \overline{\mathcal{L}^X} \to \overline{\mathcal{L}^X},$$

$$\begin{array}{rll}\text{eval}(S, t) &= t & \text{if } t \in L^s \cup \overline{L^s},\ L \in \mathcal{L},\\ \text{eval}(S, \max(x)) &= t & \text{where } x \sqsubseteq \langle \overline{s}, t\rangle \in S,\\ \text{eval}(S, \min(x)) &= \overline{s} & \text{where } x \sqsubseteq \langle \overline{s}, t\rangle \in S,\\ \text{eval}(S, \text{val}(x)) &= t & \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \in S,\\ \text{eval}(S, \text{val}(x)) &= \text{val}(x) & \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \notin S,\\ \text{eval}(S, \overline{\text{val}(x)}) &= \overline{t} & \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \in S,\\ \text{eval}(S, \overline{\text{val}(x)}) &= \overline{\text{val}(x)} & \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \notin S,\\ \text{eval}(S, \circ(t_1, \ldots, t_n)) &= \circ(\text{eval}(S, t_1), \ldots, \text{eval}(S, t_n)), &\\ \text{eval}(S, \overline{\circ}(t_1, \ldots, t_n)) &= \overline{\circ}(\text{eval}(S, t_1), \ldots, \text{eval}(S, t_n)). &\end{array}$$

Let $s, t \in L^b$. Then we further overload eval/2 and define

$$\text{eval}(S, x \sqsubseteq \langle \overline{s}, t\rangle) = x \sqsubseteq \langle \text{eval}(S, \overline{s}), \text{eval}(S, t)\rangle.$$

Definition 11 (*Constraint Propagation*). Suppose $S \in \mathcal{SS}^X$. If $c, c' \in \mathcal{C}_L^X$ and $\text{eval}(S, c) = c'$ is simple, then we say that *c is propagated* (*using S*) to c' and write $c \leadsto^S c'$. If $C \subseteq \mathcal{C}^X$ and $C' = \{c' \mid \exists c \in C \,.\, c \leadsto^S c'\}$, then we say that C *is propagated to* C' (*using* S) and write $C \leadsto^S C'$.

Note that, if C is a simple constraint store for X and $C \leadsto^S C'$, then C' is a simple constraint store for $X' \subseteq X$. Note also that, if $x \sqsubseteq \langle \overline{s}, t\rangle \in S$ where $s \neq t$, then the evaluation function eval applied to val(x) returns val(x) unchanged. Thus the indexical val(x) provides a useful tool for delaying the propagation of constraints.

Constraint stabilization is based on the intersection of simple interval constraints.

Definition 12 (*Stabilized Store*). Suppose $S \in \mathcal{S}^X$, $S' \in \mathcal{SS}^X$, and, for each $x \in X$, $S_x = \{c \in S \mid c = x \sqsubseteq r\}$. Then, if $S' = \{\bigcap_L S_x | L \in \mathcal{L}, x \in X \cap V_L\}$, we say that S' is the *stabilized store* of S and write $S \mapsto S'$.

Note that, by Definition 7, if $S_x = \emptyset$, then $\bigcap_L S_x = x \sqsubseteq \top_{R_L^b} \in \mathcal{SS}^X$.

Example 8. Suppose $r, w \in V_{\Re}$ and $i \in V_{Integer}$. Let also

$$\begin{array}{rlll} S = \{ & r \sqsubseteq \langle \overline{8.3_)}, 20.4_]\rangle, & w \sqsubseteq \langle \overline{1.2_]}, 10.5_)\rangle, & i \sqsubseteq \langle \overline{0_]}, 10_]\rangle,\\ & r \sqsubseteq \langle \overline{1.0_]}, 15.0_]\rangle, & w \sqsubseteq \langle \overline{5.6_)}, 15.3_)\rangle, & i \sqsubseteq \langle \overline{2_]}, 15_)\rangle\},\\ S' = \{ & r \sqsubseteq \langle \overline{8.3_)}, 15.0_]\rangle, & w \sqsubseteq \langle \overline{5.6_)}, 10.5_)\rangle, & i \sqsubseteq \langle \overline{2_]}, 10_]\rangle\}.\end{array}$$

Then $S \mapsto S'$. Moreover, consider the operator trunc/1 as defined in Example 3. Observe that $\text{eval}(S', \min(w)) = \overline{5.6}_{)}$ and that real values are propagated to the integer domain via the trunc operator, for example,

$$\text{eval}(S', \text{trunc}(\max(w))) = \text{trunc}(\text{eval}(S', \max(w))) = \text{trunc}(10.5_{)}) = 10_{]}.$$

Thus

$$\{r \sqsubseteq \langle \min(w), 20.4_{]}\rangle,\ i \sqsubseteq \langle \overline{\text{trunc}}(\min(w)), \text{trunc}(\max(w))\rangle\} \leadsto^{S'}$$
$$\{r \sqsubseteq \langle \overline{5.6}_{)}, 20.4_{]}\rangle,\ i \sqsubseteq \langle \overline{5}_{]}, 10_{]}\rangle\}.$$

A solution is a constraint store that cannot be reduced by means of the propagation or stabilization procedures.

Definition 13 (*Solution*). Let $C \in \wp(\mathcal{C}^X)$ be a constraint store for X. A *solution for* C is a consistent store $R \in \mathcal{SS}^X$ where,

$$C \leadsto^R C',$$
$$R \cup C' \mapsto R.$$

$Sol(C)$ denotes the set of all solutions for C. We say that $G = \text{mgs}(C)$ is a *most general solution* for C if, for all $R \in Sol(C)$, $R \preceq G$.

4.3 Monotonicity of Constraints

Our approach here has the advantage that it guarantees that the interval constraints are propagated monotonically.

Proposition 2. *Let $S_1, S_2 \in \mathcal{SS}^X$ such that S_1 and S_2 are consistent and $S_1 \preceq S_2$ and $C \in \wp(\mathcal{C}^X)$ such that*

$$C \leadsto^{S_1} C_1 \quad \textit{and} \quad S_1 \cup C_1 \mapsto S_1',$$
$$C \leadsto^{S_2} C_2 \quad \textit{and} \quad S_2 \cup C_2 \mapsto S_2'.$$

Then $S_1' \preceq S_2'$.

Example 9. Consider the definition of the operator $-$ in Example 3, when L is $\mathfrak{R}$ so that $- :: \mathfrak{R}^s \times \overline{\mathfrak{R}^s} \to \mathfrak{R}^s$. Suppose that $X = \{x, y\} \subseteq V_{\mathfrak{R}}$ and $S_1, S_2 \in \mathcal{SS}^X$,

$$S_1 = \{y \sqsubseteq \langle \overline{2.0}_{)}, 4.0_{]}\rangle,\ x ::' \mathfrak{R}\} \quad \text{and} \quad S_2 = \{y \sqsubseteq \langle \overline{1.0}_{)}, 11.0_{]}\rangle,\ x ::' \mathfrak{R}\}.$$

Then $S_1 \prec S_2$. Let

$$c_1 = x \sqsubseteq \langle \overline{0.0}_{]}, 20.0_{]} - \min(y)\rangle, \quad c_{11} = x \sqsubseteq \langle \overline{0.0}_{]}, 18.0_{)}\rangle, \quad c_{12} = x \sqsubseteq \langle \overline{0.0}_{]}, 19.0_{)}\rangle.$$

Then c_1 is an interval constraint for $\mathfrak{R}$ because $\overline{0.0}_{]} \in \overline{\mathfrak{R}^b}$ and $20.0_{]} - \min(y) \in \mathfrak{R}^b$ and hence $\langle \overline{0.0}_{]}, 20.0_{]} - \min(y)\rangle \in R^b_{\mathfrak{R}}$. Using the constraint propagation procedure for S_1 and S_2, we obtain

$$c_1 \leadsto^{S_1} c_{11}, \qquad c_1 \leadsto^{S_2} c_{12}.$$

Note that we have $c_{11} \prec c_{12}$.

Second, let

$$c_2 = x \sqsubseteq \langle \overline{0.0}_{]}, 20.0_{]} - \max(y)\rangle, \quad c_{21} = x \sqsubseteq \langle \overline{0.0}_{]}, 16.0_{]}\rangle, \quad c_{22} = x \sqsubseteq \langle \overline{0.0}_{]}, 9.0_{]}\rangle.$$

```
procedure solve($C$, $S$)
begin
  if $S$ is consistent then                                    (0)
    $C := C \cup S$;                                           (1)
    repeat
      $C \leadsto^{S} C'$;          %% Constraint Propagation   (2)
      $S' := S$;                                               (3)
      $S' \cup C' \mapsto S$;       %% Store stabilization      (4)
    until $S$ is inconsistent or $S = S'$;                     (5)
  endif;
endbegin.
```

Fig. 2. solve/2: a generic schema for interval constraint propagation.

Then c_2 is not an interval constraint in our theory. This is because, although $\overline{0.0}_{]} \in \overline{\Re^b}$ and $20.0_{]} \in \Re^b$, we have $\max(y) \notin \overline{\Re^b}$ with the consequence that, as $-$ is defined $- :: \Re^s \times \overline{\Re^s} \to \Re^s$, $20.0_{]} - \max(y) \notin \Re^b$ and hence $\langle \overline{0.0}_{]}, 20.0_{]} - \max(y) \rangle \notin R^b_{\Re}$. Applying the constraint propagation procedure to c_2 we obtain

$$c_2 \leadsto^{S_1} c_{21}, \qquad c_2 \leadsto^{S_2} c_{22}.$$

Then we have $c_{22} \prec c_{21}$. Therefore the constraint procedure applied to c_2 using the smallest constraint store S_1 derives the largest range for x. The problem is caused by the fact that if S_2 is replaced by a smaller store such as S_1, $\max(y)$ also decreases in $\Re^s$, so that the value of $20.0_{]} - \max(y)$ actually increases. Thus, the right bound of the range for x in c_2 also increases so that the upper limit for y can never be reduced.

Note that the acceptability of expressions such as c_1 and c_2 as valid constraints can be decided a priori, from the operator declarations, using standard type-checking techniques.

5. OPERATIONAL SEMANTICS

In this section, we provide an operational schema for solving the interval constraints and prove both correctness and termination properties. Note that, in this section, the main aim is to provide the basic methodology and we do not discuss possible efficiency improvements.

We continue to use L to denote any domain in $\mathcal{L}$, $X \in \wp_f(\mathcal{V}_{\mathcal{L}})$ the set of constrained variables, $\mathcal{C}^X$ the set of all interval constraints for X, and $\mathcal{SS}^X$ the set of all simple stable constraint stores for X.

5.1 Operational Schema

Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. We define here solve(C, S), an *operational schema* for computing a solution (if it exists) for $C \cup S$. This schema is shown in Figure 2.

THEOREM 1 (CORRECTNESS). *Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. If at least one solution for $C \cup S$ exists, then a terminating execution for* solve(C, S) *returns* mgs($C \cup S$) *in S. Otherwise, a terminating execution for* solve(C, S) *returns in S an inconsistent store.*

5.2 Termination

New simple constraints, created by the propagation Step (2) (see Figure 2), are added to the set of constraints before the stabilization Step (4). Thus, with infinite domains, the algorithm may not terminate since the constraints could be contracted indefinitely in the stabilization Step (4).

Example 10. Consider the operator $div2 :: \Re^s \to \Re^s$ where $div2_\Re(a) = \frac{a}{2.0}$, for any $a \in \Re$, and $div2_B$ is the identity on B. Then let C be the constraint store

$$\begin{aligned}&\{x \sqsubseteq \langle\overline{0.0_]}, 10.0_]\rangle,\ x \sqsubseteq \langle\overline{0.0_]}, div2(\max(y))\rangle,\\ &\ y \sqsubseteq \langle\overline{0.0_]}, 10.0_]\rangle,\ y \sqsubseteq \langle\overline{0.0_]}, div2(\max(x))\rangle,\\ &\ z \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle\},\end{aligned}$$

where $x, y, z \in V_\Re$. Let S_0 be the top element of the lattice $\mathcal{SS}^{\{x,y,z\}}$. Let S_i be the value of the store S at the end of the ith iteration for $i \geq 1$ of the operational schema for solve(C, S) with S_0 the initial value of S. Then, in the execution of solve(C, S), S is indefinitely reduced, that is,

$$\begin{aligned}S_0 &= \{x \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle,\ y \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle,\ z \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle\},\\ S_1 &= \{x \sqsubseteq \langle\overline{0.0_]}, 10.0_]\rangle,\ y \sqsubseteq \langle\overline{0.0_]}, 10.0_]\rangle,\ z \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle\},\\ S_2 &= \{x \sqsubseteq \langle\overline{0.0_]}, 5.0_]\rangle,\ y \sqsubseteq \langle\overline{0.0_]}, 5.0_]\rangle,\ z \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle\},\\ S_3 &= \{x \sqsubseteq \langle\overline{0.0_]}, 2.5_]\rangle,\ y \sqsubseteq \langle\overline{0.0_]}, 2.5_]\rangle,\ z \sqsubseteq \langle\overline{\bot_{\Re_]}}, \top_{\Re_]}\rangle\},\\ \ldots &\quad \ldots \quad \ldots \quad \ldots \quad \ldots .\end{aligned}$$

To force termination, we introduce the notion of *precision*.

Definition 14 (*Precision of a Constraint*). Let $\mathcal{CC}_L^X$ be the set of all consistent (and thus simple) interval constraints for L with constrained variables in X, $x \in X \cap V_L$ for any $L \in \mathcal{L}$, and $\Re\mathcal{I}$ denote the lexicographic product $(\Re^+, Integer)$ where $\Re^+$ is the (lifted) domain of nonnegative reals. Then we define

$$\begin{aligned}&precision_L :: \mathcal{CC}_L^X \to \Re\mathcal{I},\\ &precision_L(x \sqsubseteq \langle\overline{a_b}, c_d\rangle) = (\hat{a} \diamond_L c, b \diamond_B d),\end{aligned}$$

where $\diamond_L :: \{(\hat{a}, c) \mid a, c \in L, a \preceq c\} \to \Re^+$ is a (system or user defined) strict monotonic function, and $\diamond_B :: B \times B \to \{0, 1, 2\}$ is the strict monotonic function

$$\text{‘]’} \diamond_B \text{‘]’} \stackrel{\text{def}}{=} 2, \qquad \text{‘]’} \diamond_B \text{‘)’} \stackrel{\text{def}}{=} 1, \qquad \text{‘)’} \diamond_B \text{‘]’} \stackrel{\text{def}}{=} 1, \qquad \text{‘)’} \diamond_B \text{‘)’} \stackrel{\text{def}}{=} 0.$$

Observe that $precision_L$ is defined only on consistent constraints and thus the function $\diamond_L$ only needs to be defined when its first argument is less than or equal to the second. This function must be defined for each computation domain including any fictitious top or bottom elements.

Example 11. Assume that $\Re^2 = \langle\Re, \Re\rangle$ and suppose that $i_1, i_2 \in Integer$, $r_1, r_2, w_1, w_2 \in \Re$, and $s_1, s_2 \in Set\ Integer$ where $i_1 \preceq i_2$, $r_1 \preceq r_2$, $w_1 \preceq w_2$, and

$s_1 \preceq s_2$. Then

$$\begin{aligned}
\widehat{i_1} \diamond_{Integer} i_2 &= i_2 - i_1,\\
\widehat{r_1} \diamond_{\Re} r_2 &= r_2 - r_1,\\
\widehat{\langle r_1, w_1\rangle} \diamond_{\Re^2} \langle r_2, w_2\rangle &= +\sqrt{(r_2 - r_1)^2 + (w_2 - w_1)^2},\\
\widehat{s_1} \diamond_{Set\ Integer} s_2 &= \#s_2 - \#s_1.
\end{aligned}$$

Assume that $i \in V_{Integer}$, $r \in V_{\Re}$, $y \in V_{\Re^2}$ and $s \in V_{Set\ Integer}$. Then

$$\begin{aligned}
&precision_{Integer}(i \sqsubseteq \langle \overline{1_]}, 4_]\rangle) = (3.0, 2),\\
&precision_{\Re}(r \sqsubseteq \langle \overline{3.5_)}, 5.7_)\rangle) = (2.2, 0),\\
&precision_{\Re^2}(y \sqsubseteq \langle \overline{(2.0, 3.0)_]}, (3.4, 5.6)_]\rangle) = (2.95, 2),\\
&precision_{Set\ Integer}(s \sqsubseteq \langle \overline{\{\}_]}, \{3, 4, 5\}_)\rangle) = (3.0, 1).
\end{aligned}$$

Note that the binary operators used in this example, that is, $-$ and $+$ as well as the unary operators # and "square" need to be defined for both the lifted bounds. The unary operator "square root" must be defined just for the lifted upper bound.

By defining a computable[2] bound $\delta \in \Re\mathcal{I}$ (user- or system-defined), we can check if the precision of the simple constraints in a store S are reduced by a significant amount in the stabilization process (Step 4 in the operational schema for solve(C, S)). 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. This "solution" is an approximation to the concept of solution shown in Definition 13.

Definition 15. Let $S, S' \in \mathcal{SS}^X$ be two consistent stores where c_x, c'_x denote the consistent constraints for $x \in X$ in S and S', respectively. Then, we define

$$no\_difference_\delta(S', S) \iff \forall L \in \mathcal{L} : \forall x \in X \cap V_L :$$
$$precision_L(c'_x) - precision_L(c_x) \preceq \delta,$$

where $(a_1, a_2) - (b_1, b_2) = (a_1 - b_1, a_2 - b_2)$ and, for $L \in \{Integer, \Re^+\}$, $x - y$ is defined as usual over any $x, y \in L$ and also $\top_L - x = \top_L$ for any $x \in L \cup \{\bot_L\}$ and $\top_L - \top_L = 0$.

We define a new operational schema for solve$_\varepsilon(C, S)$ which, apart from Step (5) in Figure 2, is the same as the solve(C, S) schema. This step is replaced by:

(5*) *until* S *is inconsistent or* $no\_difference_{(\varepsilon,0)}(S', S)$ *holds.*

THEOREM 2 (TERMINATION). *Let* $C \in \wp_f(\mathcal{C}^X)$ *and* $S \in \mathcal{SS}^X$. *If* $\varepsilon > 0.0$ *then the operational schema for* solve$_\varepsilon(C, S)$ *terminates.*

Definition 16 (*Approximate Solution*). Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. Let also R be a solution for $C \cup S$ and $\delta \in \Re\mathcal{I}$. Then R' is an *approximate solution via* δ for $C \cup S$ if $R \preceq R'$ and $no\_difference_\delta(R', R)$ holds.

[2]That is, representable in the machine which is being used—the computation machine.

The number of iterations of the operational schema depends, for infinite domains, on the value of ε. In these cases, the final solution for $\text{solve}_\varepsilon(C, S)$ is an approximate solution for $C \cup S$.

Theorem 3 (Extended Correctness). *Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. If at least one solution R for $C \cup S$ exists, a terminating execution of the operational schema for* $\text{solve}_\varepsilon(C, S)$ *computes in S an approximate solution for $C \cup S$.*

The next result shows that the approximate solution is dependent on the value of ε in the sense that lower ε is, closer the approximate solution to the solution is.

Theorem 4. *Let R be a solution for $C \cup S$ where $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. Suppose that S_{ε_1} and S_{ε_2} are the approximate solutions computed by the operational schema for* $\text{solve}_{\varepsilon_1}(C, S)$ *and* $\text{solve}_{\varepsilon_2}(C, S)$, *respectively. Then, if* $0.0 \leq \varepsilon_1 \leq \varepsilon_2$,

$$R \preceq S_{\varepsilon_1} \preceq S_{\varepsilon_2}.$$

The *precision* map and the bound ε allow direct and transparent control over the accuracy of the results. For example, we could define $\varepsilon = 10^{-8}$ for reals. Together, the *precision* map and the bound ε provide a concept of graded solutions to a constraint problem as well as a concept of distance to the correct solution: the higher the bound ε, then the further away (from the correct solution) is the approximate solution. The technique for ensuring termination described here is the first (to our knowledge) fully generic proposal. Note that Sidebottom and Havens [1992] described a similar approach for the real numbers which associates a precision parameter with the real domain and limits the number of times that the domain of a variable may be refined.

6. APPLICATIONS OF OUR FRAMEWORK

The *Integer*, $\Re$, *Bool*, and *Set L* (for any domain $L \in \mathcal{L}$) domains have been used throughout the paper to illustrate the concepts as they were defined. In this section, we provide further examples of computation domains, various combinations of the domains using lattice combinators, and a simple extension of the framework to allow for high level constraints.

6.1 Computation Domains: More Examples

Strings. Walinsky [1989] described CLP(Σ^*), a CLP language that incorporates strings in logic programming by means of membership constraints on finite strings of characters, and indicated the usefulness of this language in a number of diverse areas such as text processing applications or security maintenance in information systems. Our framework can be used to emulate CLP(Σ^*) as follows. Consider the domain *String*, the set of all sequences (possibly infinite) of characters together with $\top_{String}$. Assume also the ordering for *String* is defined such that for $a_1, a_2 \in String$, $a_1 \preceq a_2$ if and only if a_1 is a prefix (finite initial substring) of a_2. Then, letting $\perp_{String}$ be the empty string, we have the lattice

$$(String, \preceq, \mathrm{glb}_{String}, \mathrm{lub}_{String}, \bot_{String}, \top_{String}).$$

Note that $\mathrm{glb}_{String}(a_1, a_2)$ is the largest common prefix of a_1, a_2 and $\mathrm{lub}_{String}(a_1, a_2)$ is $\top_{String}$ if $a_1 \not\sim a_2$, a_2 if $a_1 \preceq a_2$, and a_1 if $a_2 \preceq a_1$.

Suppose that $+\!\!+/2$ is a constraint operator that concatenates two strings. Let $a, a', a'' \in V_{String}$ and

$$\begin{aligned} S_1 &= \{a' \sqsubseteq \langle \bot_{String_]}, \top_{String_]} \rangle,\ a \sqsubseteq \langle \overline{\text{‘abc’}_]}\ \overline{+\!\!+}\ \min(a'), \top_{String_]} \rangle\}, \\ S_2 &= \{a' \sqsubseteq \langle \bot_{String_]}, \top_{String_]} \rangle,\ a'' \sqsubseteq \langle \bot_{String_]}, \top_{String_]} \rangle, \\ &\qquad a \sqsubseteq \langle\ \overline{\mathrm{val}(a')}\ \overline{+\!\!+}\ \overline{\text{‘abc’}_]}\ \overline{+\!\!+}\ \overline{\mathrm{val}(a'')}, \mathrm{val}(a')\ +\!\!+\ \text{‘abc’}_]\ +\!\!+\ \mathrm{val}(a'') \rangle\}. \end{aligned}$$

Then, S_1 and S_2 constrains, respectively, $a \in V_{String}$ to be any string with prefix ‘abc’ and any string containing the string ‘abc’.

Nonnegative integers ordered by division. Let $\mathcal{N}_d$ denote the set of nonnegative integers partially ordered by division: for all $n, m \in \mathcal{N}_d$, $m \preceq n$ if and only if $\exists k \in \mathcal{N}_d$ such that $km = n$ (that is, m divides n). Then

$$(\mathcal{N}_d, \preceq, gcd, lcm, 1, 0)$$

is a lattice where gcd denotes the greatest common divisor and lcm the least common multiple. Thus with $\mathcal{N}_d$ as the computation domain, we have

$$x \sqsubseteq \langle \overline{2_]}, 24_] \rangle \cap_L x \sqsubseteq \langle \overline{3_]}, 36_] \rangle = x \sqsubseteq \langle \overline{6_]}, 12_] \rangle.$$

Numeric intervals. We consider $Interv$ as the domain of the numeric intervals. We define $it_1 \preceq it_2$ if and only if $it_1 \subseteq it_2$ (i.e., it_1 is a subinterval of it_2). Thus glb_{Interv} and lub_{Interv} are the intersection and union of intervals, respectively. Our framework solves constraints for the $Interv$ computation domain as follows:

$$i \sqsubseteq \langle \overline{[5,6]_]}, [2,10)_] \rangle \cap_L i \sqsubseteq \langle \overline{(7,9]_]}, [4,15]_] \rangle = i \sqsubseteq \langle \overline{[5,6] \cup (7,9]_]}, [4,10)_] \rangle.$$

Note that $\overline{[5,6] \cup (7,9]_]} \in \overline{Interv^s}$ whereas $[4,10)_] \in Interv^s$.

6.2 Combinations of Domains

Our lattice-based framework allows for new computation domains to be constructed from previously defined domains. Here we give examples which use well-known lattices combinators.

Product of domains. As already observed, the direct and lexicographic products of lattices are lattices. For example, consider the lattice $Integer$.

(1) A point in a plane may be defined by its Cartesian coordinates using the direct product $Point = \langle Integer, Integer \rangle$.
(2) A rectangle can be defined by two points in a plane: its lower left corner and its upper right corner. Let $Rect$ be the direct product $\langle Point, Point \rangle$.

Interval constraints can be declared directly on these domains. For example, consider $re \in V_{Rect}$; then

$$re \sqsubseteq \langle \overline{\langle\langle 2,2\rangle, \langle 5,5\rangle\rangle}_{]}, \langle\langle 4,4\rangle, \langle 7,7\rangle\rangle_{]}\rangle$$

constrains the rectangle re to have its lower left corner in the plane $\langle 2,2\rangle \times \langle 4,4\rangle$ and its upper right corner in the plane $\langle 5,5\rangle \times \langle 7,7\rangle$. Thus the rectangle $\langle\langle 3,3\rangle, \langle 6,6\rangle\rangle$ satisfies this constraint.

Sum of domains. The linear sum of $n > 1$ lattices is also a lattice.

Definition 17 (*Linear Sum*). Suppose that $L_1, \ldots, L_n$ are lattices. Then their *linear sum* $L_1 \uplus \cdots \uplus L_n$ is the lattice L_S where

(1) $L_S = L_1 \cup \cdots \cup L_n$;

(2) the ordering relation $\preceq$ is defined:

$$x \preceq y \iff \begin{cases} x, y \in L_i \text{ and } x \preceq y \text{ or} \\ x \in L_i,\ y \in L_j \text{ and } i \prec j; \end{cases}$$

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

$$\begin{aligned} \mathrm{glb}_{L_S}(x,y) &= \mathrm{glb}_{L_i}(x,y) \text{ and } \mathrm{lub}_{L_S}(x,y) = \mathrm{lub}_{L_i}(x,y) \quad \text{if } x, y \in L_i, \\ \mathrm{glb}_{L_S}(x,y) &= x \text{ and } \mathrm{lub}_{L_S}(x,y) = y \quad \text{if } x \in L_i,\ y \in L_j \text{ and } i \prec j, \\ \mathrm{glb}_{L_S}(x,y) &= y \text{ and } \mathrm{lub}_{L_S}(x,y) = x \quad \text{if } x \in L_i,\ y \in L_j \text{ and } j \prec i; \end{aligned}$$

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

As an example, consider the lattice *AtoF* containing all the (uppercase) alphabetic characters between 'A' and 'F' with the usual alphabetical ordering and the lattice *0to9* containing the numeric characters from '0' to '9' with the ordering '0' < '1' < $\cdots$ < '8' < '9'. Then the lattice of hexadecimal digits can be defined as the lattice *0to9* $\uplus$ *AtoF*. Now, it is possible to constrain variables to have values in such a domain. For example a code of four hexadecimal digits can be initially represented by four variables $h_1h_2h_3h_4$ that are constrained by a type constraint as h_1, h_2, h_3, h_4 ::′ *0to9* $\uplus$ *AtoF* (note that this is equivalent to the constraints $h_1 \sqsubseteq \langle\overline{\text{'0'}}_{]}, \text{'F'}_{]}\rangle$, $h_2 \sqsubseteq \langle\overline{\text{'0'}}_{]}, \text{'F'}_{]}\rangle$, $h_3 \sqsubseteq \langle\overline{\text{'0'}}_{]}, \text{'F'}_{]}\rangle$, and $h_4 \sqsubseteq \langle\overline{\text{'0'}}_{]}, \text{'F'}_{]}\rangle$).

6.3 High-Level Constraints

A constraint operator can provide a useful one-way channel of communication by allowing values in the computation domains for its arguments to be propagated to the computation domain in its range. Here we define *high-level constraints* by means of a generic relation that enables the propagation of information between domains in any direction allowing for full cooperation between the solvers for these domains. So as to distinguish the interval constraints defined and studied in previous sections from the high-level constraints defined here, we call constraints of the form $x \sqsubseteq r$ *primitive constraints*.

Definition 18 (*High-Level Constraint*). Suppose that $\mathcal{L}' = \{L_1, \ldots, L_m\} \subseteq \mathcal{L}$. Then $q :: L_1 \times \cdots \times L_m$ is called an *m-ary constraint relation* for $\mathcal{L}'$. Suppose $x_1 \in V_{L_1}, \ldots, x_m \in V_{L_m}$, and $\{c_1, \ldots, c_n\}$ is a constraint store with constrained variables $X \supseteq \{x_1, \dots, x_m\}$. Then

$$q(x_1, \dots, x_m) \Leftrightarrow c_1, \dots, c_n$$

is called a *high-level constraint* over $\mathcal{L}'$.

Note that our high-level constraints are similar to clp(FD) [Codognet and Diaz 1996a]. However, unlike clp(FD), our framework also allows for the definition of both generic and overloaded constraints: A high-level constraint is *generic* for arguments $i_1, \dots, i_j$ $(1 \leq i_1 < \cdots < i_j \leq m)$ if its definition is independent of the choice of domains $L_{i_1}, \dots, L_{i_j}$ in $\mathcal{L}$; A constraint is *overloaded* for arguments $i_1, \dots, i_j$ if it is defined for any $L_{i_1}, \dots, L_{i_j}$ in $\mathcal{L}_1$ where $\mathcal{L}_1 \subset \mathcal{L}$ and $\#(\mathcal{L}_1) > 1$.

Example 12. Consider the following high-level "less-or-equal" constraint:

$$\begin{aligned} x \leq y \;\Leftrightarrow\; x &\sqsubseteq \langle \overline{\bot_{L_\rceil}}, \max(y)\rangle, \\ y &\sqsubseteq \langle \min(x), \top_{L_\rceil}\rangle. \end{aligned} \tag{3}$$

Then this is generic for both arguments of $\leq$ as each $L \in \mathcal{L}$ has (possible lifted) top and bottom elements. Also, consider the definition of the operators $+$ and $-$ shown in Example 3 and the following definition of a `plus`/3 constraint:

$$\begin{aligned} \texttt{plus}(x, y, z) \Leftrightarrow x &\sqsubseteq \langle \min(z) \overline{-} \max(y), \max(z) - \min(y)\rangle, \\ y &\sqsubseteq \langle \min(z) \overline{-} \max(x), \max(z) - \min(x)\rangle, \\ z &\sqsubseteq \langle \min(x) \overline{+} \min(y), \max(x) + \max(y)\rangle. \end{aligned}$$

This constraint is overloaded since it is valid for any domain L in which operators $+_L$ and $-_L$ are defined. For example, a call $plus(x, y, z)$, where $x, y, z \in V_L$ for $L \in \{Integer, \Re\}$, means $x = y + z$ whereas, for $L = Set\ L'$ and $L' \in \mathcal{L}$, it means $(x \cup y = z) \wedge (x \cap y = \emptyset)$.

Note that in an implementation, if C is a constraint store containing a high-level constraint $c \Leftrightarrow c_1, \dots c_n$, then we replace c (in C) by $\{c_1, \dots, c_n\}$. This has to be repeated until C contains no high-level constraints. Of course, termination of this is not guaranteed and would depend on the definitions of the high-level constraints. Once C is simplified to a set of primitive constraints, constraint propagation and constraint stabilization can be executed as usual.

7. A CLP LANGUAGE FOR LATTICES

In this section, we describe $clp(\mathcal{L})$ (Constraint Logic Programming on any set $\mathcal{L}$ of lattices),[3] a language that we have implemented to validate the feasibility of the framework. We also provide a simple scheduling example to illustrate how $clp(\mathcal{L})$ may be used to solve a constraint problem.

[3]Sources, user manual, and a number of $clp(\mathcal{L})$ examples are available from http://www.lcc.uma.es/~afdez/generic. Note that all the examples shown in this paper were tested on this prototype.

7.1 The *clp* ($\mathcal{L}$) Language

This language is based on standard Prolog [ISO/IEC 1995], with some extra declarations for specifying both new domains and constraint operators. Thus the set of computation domains $\mathcal{L}$ over which the constraints may be defined is the set of all system and user-declared domains in the program. The advantage of our theoretical approach with respect to other generic approaches (see Section 9) is that it can be implemented directly from the theory. The interval constraints such as $x \sqsubseteq r$ are expressed in $clp(\mathcal{L})$ as expressions of the form X `isin` R and can occur anywhere in the bodies of clauses. A simple range $\langle \overline{a_b}, c_d \rangle$ is expressed as an expression $(a, \text{bracket}(b))..(c, \text{bracket}(d))$ where bracket(']') = `close` and bracket(')') = `open` and `open` and `close` are reserved words in the system, for example, $\langle \overline{1.2_)}, 4.5_] \rangle$ is denoted as (1.2, `open`)..(4.5, `close`).

New domains can be defined by the user with a *lattice declaration*: the predicate `lattice`/2 identifies the elements belonging to the new lattice; and the predicates `lt`/3, `glb`/4, and `lub`/4 define, respectively, the ordering relation, glb, and lub for the domain. That is to say,

—`lattice`(D, E) is true if $E \in D$;
—`lt`(D, X, Y) is true if $X < Y$ in D;
—`glb`(D, X, Y, Z) is true if $Z = \text{glb}_D(X, Y)$;
—`lub`(D, X, Y, Z) is true if $Z = \text{lub}_D(X, Y)$.

The current prototype implementation of $clp(\mathcal{L})$ provides some predefined predicates for the combination of domains (e.g., `product_Direct`/3, `linear_sum`/3, and `product_Lexicographic`/3). However, new lattice combinators can be easily implemented in a declarative way via domain declarations. As examples, Figure 3 shows how to declare the set domain (defined as a list with the inclusion as ordering) and the direct product domain `reint_point` = ⟨`real`, `integer`⟩ (assuming `integer` and `real` are system-defined domains *Integer* and $\Re$, respectively).

The language allows the declaration of both unary and binary operators by means of the predicates `declara`/3 or `declara`/4. Let L, L_1, L_2 be (user or system) (not necessarily distinct) computation domains. Then

—`declara`(Op, L_1, L) specifies the unary operator $Op :: L_1^s \rightarrow L^s$,
—`declara`(Op, L_1, L_2, L) specifies the binary operator $Op :: L_1^s \times L_2^s \rightarrow L^s$.

If L_1 or L_2 is replaced in the above by `mirror`(L_1) or `mirror`(L_2), then the domains L_1^s or L_2^s are replaced by the mirrors $\overline{L_1^s}$ or $\overline{L_2^s}$. Figure 4 shows the definition of the operators[4] $+$ and $-$ of Example 3 over the domains of integers, reals, sets, and `reint_point`. Observe that these operators are defined on the bracket domain and each of the computation domains (see Definition 3). Particularly, for the set domain, :+: and :−: are defined to be the usual union and difference of sets, respectively.

The implementation manages a single constraint store which contains all the (user or system) defined (primitive) interval constraints for domains in $\mathcal{L}$. As in

[4]By syntax conventions, $clp(\mathcal{L})$ operators begin and end with colons.

```
                              %--- LATTICE DECLARATION
lattice(set,Ele) :- is_list(Ele).
                              %--- SET ORDERING
lt(set,S1,S2) :- lattice(set,S1), lattice(set,S2),
                 notequal(S1,S2), include_in(S1,S2).
                              %--- GLB AND LUB
glb(set,X,Y,Z) :- lattice(set,X), lattice(set,Y), !, intersection(X,Y,Z).
lub(set,X,Y,Z) :- lattice(set,X), lattice(set,Y), !, union(X,Y,Z).
                              %--- ADDITIONAL PREDICATES
include_in([],_S).
include_in([X|Y],S) :- member(X,S), include_in(Y,S).

equal(S1,S2) :- include_in(S1,S2), include_in(S2,S1).

notequal(S1,S2) :- not(equal(S1,S2)).

union(S1,S2,S3) :- lattice(set,S1), lattice(set,S2),!,
                 append(S1,S2,S), remove_duplicates(S,S3).

intersection([],S2,[]) :- lattice(set,S2),!.
intersection([X|Y],S2,[X|S]) :- lattice(set,S2), member(X,S2), !,
                 intersection(Y,S2,S).
intersection([X|Y],S2,S) :- lattice(set,S2), not(member(X,S2)), !,
                 intersection(Y,S2,S).

                              %--- DECLARATION OF COMBINED DOMAIN
lattice(reint_point,(A,B)) :- lattice(real,A), lattice(integer,B).
lt(reint_point,(A,B),(C,D)) :- lt(real,A,C), lt(integer,B,D).
glb(reint_point,(A,B),(C,D)) :- glb(real,A,C), glb(integer,B,D).
lub(reint_point,(A,B),(C,D)) :- lub(real,A,C), lub(integer,B,D).
```

Fig. 3. Lattice declarations: *Set L* and reint_point.

```
    %--- Operator declarations
declara(:+:,L,L,L).
declara(:-:,L,mirror(L),L).
    %--- Definition of :+: and :-: on the bracket domain.
:+:(close,close,close).    :+:(open,_,open).         :+:(_,open,open).
:-:(B,B,close).            :-:(B1,B2,open):-B1\==B2.
    %--- Definition of :+: on the integer, real, set and reint_point domains
:+:(E1,E2,E3) :- lattice(integer,E1), lattice(integer,E2), E3 is E1+E2.
:+:(E1,E2,E3) :- lattice(real,E1), lattice(real,E2), E3 is E1+E2.
:+:(E1,E2,E3) :- lattice(set,E1), lattice(set,E2), union(E1,E2,E3).
:+:((A,B),(C,D),(E,F)) :- lattice(reint_point,(A,B)),
                lattice(reint_point,(C,D)), E is  A + C, F is B + D.
    %--- Definition of :-: on the integer, real, set and reint_point domains
:-:(E1,E2,E3) :- lattice(integer,E1), lattice(integer,E2), E3 isE1-E2.
:-:(E1,E2,E3) :- lattice(real,E1), lattice(real,E2), E3 is E1-E2.
:-:(E1,E2,E3) :- lattice(set,E1), lattice(set,E2), difference(E1,E2,E3).
:-:((A,B),(C,D),(E,F)) :- lattice(reint_point,(A,B)),
                lattice(reint_point,(C,D)), E is  A - C, F is B - D.
    %--- ADDITIONAL PREDICATES
difference([],_,[]).
difference([X|S1],S2,[X|S3]) :- not(member(X,S2)), !, difference(S1,S2,S3).
difference([X|S1],S2,S3) :- member(X,S2), !, difference(S1,S2,S3).
```

Fig. 4. Operator declarations in $clp(\mathcal{L})$.

Prolog, the resolution mechanism of the $clp(\mathcal{L})$ language is LD-resolution with a special procedure for binding constrained variables. That is, when binding a variable X in a domain L to any term t distinct from X, the unification step is

—if t is an unbound variable Y, then Y is bound to X;

—if t is a term $l \in L$, then the constraint "X isin $(l, \texttt{close})..(l, \texttt{close})$" is added to the store; or

—if t is a constrained variable $Y \in V_L$, the constraints "X isin $\min(Y)..\max(Y)$" and "Y isin $\min(X)..\max(X)$" are added to the store.

The step returns a fail, called *domain fail*, if t is either a constrained variable in $V_{L'}$ or term in L' and $L' \neq L$. Also, using the operator declarations, the prototype identifies the nonmonotonic constraints (i.e., constraints not contributing to the solution—see Section 4.3).

Our prototype implementation of $clp(\mathcal{L})$ [Fernández 2000] is built on the SICStus 3#7 Prolog platform [Carlsson et al. 1997]. Constraint consistency, store stabilization, and constraint propagation are implemented using the constraint handling rules (CHRs) [Frühwirth 1998] that are part of a SICStus library. The CHRs are very appropriate since they are solved prior to the resolution step of the standard logical engine. The current $clp(\mathcal{L})$ implementation provides predefined Boolean constraints such as `and`/3, `or`/3, `xor`/3, `equiv`/3, and `not`/2 among others; symbolic constraints such as `at_least_one`/1, `at_most_one`/1, and `only_one`/1; arithmetic constraints such as `plus`/3, `diff`/3, `divide`/3, and `times`/3 as well as generic arithmetic constraints such as $=/2$, $\neq/2$, $>/2$, $\geq/2$, $<$ $/2$, and $\leq/2$ defined on usual numerical domains and on combined domains. As $clp(\mathcal{L})$ is implemented in SICStus, the prototype supports many system predicates provided by SICStus.

Example 13. Figure 5 shows how to code the high-level constraints in Example 12 as well as how the overloaded `plus` constraint is used.[5]

7.2 An Overloaded Generic Scheduling Problem

This example scheduling problem illustrates the generic power of our solver[6] where the tasks are represented by terms of the form $\texttt{Task}(S, D)$; S is the start time and D is the duration. The high-level constraint `into(Task, SuperTask)` is true if the interval for `SuperTask` contains the interval for `Task`; `noOverlap(Task, Tasks)` is true if `Task` overlaps with no elements in

[5]In $clp(\mathcal{L})$, $x :: {'}L$ denotes a type constraint for x in L (see Definition 7) and $[x_1, \dots, x_n] :: {'}L$ is equivalent to $x_1 :: {'}L, \dots, x_n :: {'}L$. `bottom` and `top` are reserved words denoting fictitious bottom and top elements for any lattice. Observe also that the constraint `T isin` (3, `close`)..(11, `open`) was reduced to the constraint `T isin` (3, `close`)..(10, `close`) by applying the equivalence rules for discrete domains we will describe in Section 8. Note that the prompt is `clp(L) >`.

[6]This example is a generalization of a program proposed in Sidebottom and Havens [1992] formulated for the real domain and using some of the relations on temporal intervals described in Allen [1983].

```
                  %%% HIGH LEVEL CONSTRAINTS
        plus(X,Y,Z) :- X isin ((min Z):-:(max Y))..((max Z):-:(min Y)),
                       Y isin ((min Z):-:(max X))..((max Z):-:(min X)),
                       Z isin ((min X):+:(min Y))..((max X):+:(max Y)).

        X <=: Y :- X isin (bottom,close)..(max Y),
                   Y isin (min X)..(top,close).

GOAL:
clp(L) >[X,Y,Z]::'real, [V,W,T]::'integer,
  [C1,C2,C3]::'set, [P1,P2,P3]::'reint_point,
  Z isin (1.0,close)..(4.0,close),          Y isin (0.0,open)..(90.0,close),
  V isin (1,close)..(2,close),              W isin (2,close)..(9,open),
  C1 isin ([1],close)..([1,2,3],close),     C2 isin ([4],close)..([4,7],close),
  P1 isin ((0.5,0),close)..((1.8,2),close), P2 isin ((1.2,3),close)..((2.1,9),close),
  plus(X,Y,Z), plus(V,W,T), plus(C1,C2,C3), plus(P1,P2,P3).

SOLUTION:
X isin (-89.0,close)..(4.0,open),           T isin (3,close)..(10,close),
C3 isin ([1,4],close)..([1,2,3,4,7],close), P3 isin ((1.7,3),close)..((3.9,11),close).
```

Fig. 5. Using the overloaded constraint *plus*/3.

Tasks; schedule(Tasks, SuperTask) is true if every task in Tasks is in SuperTask and no pair overlaps.

Figure 6 shows code for solving this problem and provides an application of the generic and overloading capabilities of *clp*($\mathcal{L}$). (Observe that the constraints into/2, noOverlap/2, and schedule/2 are overloaded since they are defined in terms of both the overloaded constraint plus/3 and the generic constraint <=:/2 that are already defined in Figure 5.) Two of the instances use the FD and real domains. The third instance is more interesting and uses the combined domain reint_point as declared in preceding section. Suppose there are two processes *p1 = (S1,D1)* and *p2 = (S2,D2)* where *p1* must be executed on a machine A in real time and *p2* on a machine B in discrete time. Then, in this instance a task consists in the resolution of both processes and can be represented as the term Task(($S1$, $S2$), ($D1$, $D2$)). The solution can be interpreted as follows: process *p1* has to begin its execution in machine A during the interval [3.75, 5.125] and, in this case, process *p2* has to start its execution, in machine B, during the interval [4, 5] (i.e., in the fourth or fifth unit of time in machine B). Alternatively, *p1* can begin its execution during the interval [0.7, 1.875] and, then, *p2* has to start during the interval [1, 2] (i.e., in the first or second unit of time in machine B). Solutions for each of the instances are graphically illustrated in Figure 7 where black dots mark the solution set.

8. OPTIMIZATIONS

This section discusses improvements that are being made to the *clp*($\mathcal{L}$) system.

8.1 Discrete Domains

Suppose that L is a discrete domain. Then we can identify equivalent elements of L^s and hence the interval domain R^s_L by introducing the following *equivalence*

```
PROGRAM:
  into(task(S1,D1),task(S2,D2)):- S2 <=: S1, plus(S1,D1,SD1),
                                  plus(S2,D2,SD2), SD1 <=: SD2.
  noOverlap(_,[]).
  noOverlap(task(S1,D1),[task(S2,D2)|Tasks]):-
                 ((plus(S1,D1,SD1), SD1 <=: S2)
                 ;
                  (plus(S2,D2,SD2), SD2 <=: S1)),
                 noOverlap(task(S1,D1),Tasks).
  schedule([],_).
  schedule([Task|Tasks],Supertask):- into(Task,Supertask),
                 noOverlap(Task,Tasks),
                 schedule(Tasks,Supertask).

INSTANCE ON clp(FD)
  clp(L) > schedule([task(0,1),task(3,1),task(S,1)],task(0,6)).

  solution: S isin (1,close)..(2,close) or S isin (4,close)..(5,close).

INSTANCE ON clp(Real)
  clp(L) > schedule([task(0.0,0.7),task(2.75,1.0),task(S,0.875)], task(0.0,6.0)).

  solution: S isin (0.7,close)..(1.875,close) or S isin (3.75,close)..(5.125,close)

INSTANCE ON clp(Real x FD)
  clp(L) > schedule([task((0.0,0),(0.7,1)),task((2.75,3),(1.0,1)),task(S,(0.875,1))],
                                                         task((0.0,0),(6.0,6))).

  solution: S isin ((3.75,4),close)..((5.125,5),close) or
            S isin ((0.7,1),close)..((1.875,2),close).
```

Fig. 6. An overloaded scheduling program.

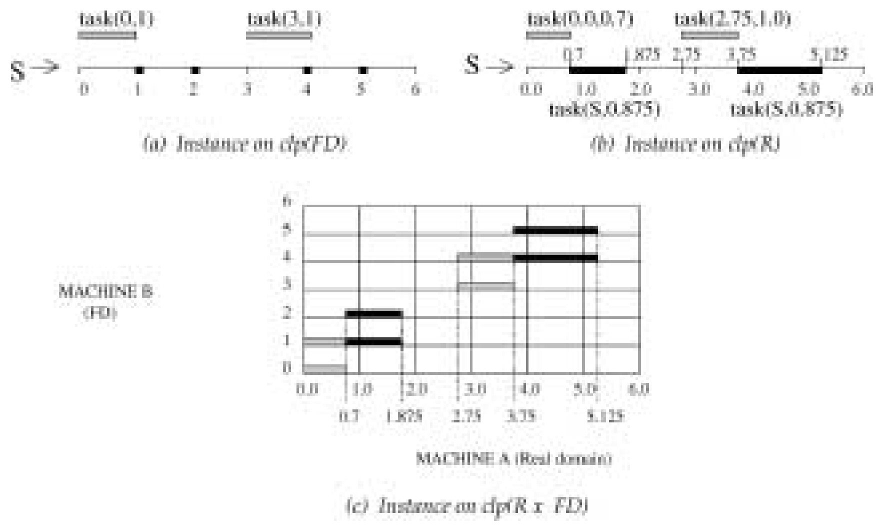

Fig. 7. Solving a scheduling problem.

rule: for any $a \in L$ for which the *immediate predecessor* pre(a) is defined and is unique,

$$a_) \equiv \text{pre}(a)_] \quad \text{in } L^s.$$

By the duality principle of lattices we also have the dual rule: for any $a \in L$ for which the *immediate successor* succ(a) is defined and is unique,

$$\overline{a_)} \equiv \overline{\text{succ}(a)_]} \quad \text{in } \overline{L^s}.$$

When the interval domain L^s is constructed from a discrete domain L, then pre and succ provide a canonical form for L^s and $\overline{L^s}$ (and hence for R^s_{L}) where the bracket ')' is eliminated in favor of the bracket ']'.

Example 14. For the *Integer* domain, pre(i) $= i - 1$ for any $i \in \textit{Integer}$ so that, for instance, $3_) \equiv 2_]$ and $\langle \overline{1_]}, 3_) \rangle \equiv \langle \overline{1_]}, 2_] \rangle$. Suppose $L = \{0, 1, 2, 3\}$ is a lattice where $0 < 1 < 2 < 3$. Then,

$$L^s = \{0_), 0_], 1_), 1_], 2_), 2_], 3_), 3_]\} \equiv \{0_), 0_], 1_], 2_], 3_]\},$$
$$\overline{L^s} = \{\overline{3_)}, \overline{3_]}, \overline{2_)}, \overline{2]}, \overline{1_)}, \overline{1_]}, \overline{0_)}, \overline{0_]}\} \equiv \{\overline{3_)}, \overline{3_]}, \overline{2_]}, \overline{1_]}, \overline{0_]}\}.$$

Similarly, with the *Bool* domain, pre(*true*) $=$ *false* so that $\textit{true}_) \equiv \textit{false}_]$ and $\langle \overline{\textit{false}_]}, \textit{true}_) \rangle \equiv \langle \overline{\textit{false}_]}, \textit{false}_] \rangle$.

With these rules for discrete domains, more inconsistencies can be detected.

Example 15. Consider again the domain L in Example 6. Suppose L is discrete and that succ(a) $= c$ and succ(c) $= b$. Then the ranges $\langle \overline{a_)}, c_) \rangle \in R^s_{\underline{L}}$ and $\langle \overline{c_)}, b_) \rangle$ are inconsistent since they are equivalent to ranges $\langle \overline{c_]}, a_] \rangle$ and $\langle \overline{b_]}, c_] \rangle$, respectively. These are the circled nodes in Figure 1.

8.2 Nonlinear Constraints and Floating Point Arithmetic

For the real domain, many constraint systems provide both a linear and a nonlinear solver. As the linear solver is the most efficient, this should be used whenever the constraints are linear. Although our solver does not provide direct support for solving nonlinear numeric equations, nonlinear constraints can be solved in $clp(\mathcal{L})$ by defining appropriate constraint operators.[7]

To see this, consider $\Re^+$, the domain of nonnegative reals, and a constraint such as '$x * y = z$'/3. The main problems occur if it tries to evaluate z/x when $x = 0.0$ or z/y when $y = 0.0$. Of course, this constraint can be delayed until either x or y is ground and then check whether or not the ground term is 0.0. However, more propagation can be obtained by defining '$x * y = z$'/3 as a high-level constraint:

$$\begin{aligned} x * y = z \Leftrightarrow\ & x \sqsubseteq \langle \min(z) \,\overline{/_l}\, \max(y), \max(z) \,/_r\, \min(y) \rangle, \\ & y \sqsubseteq \langle \min(z) \,\overline{/_l}\, \max(x), \max(z) \,/_r\, \min(x) \rangle, \\ & z \sqsubseteq \langle \min(x) \,\overline{*}\, \min(y), \max(x) \,*\, \max(y) \rangle, \end{aligned}$$

[7]This is a generalization of the method proposed in Codognet and Diaz [1996a] for the clp(FD) system, and we use the same example in this paper.

where $*$, $/_l$, $/_r$ are declared in $\Re^{+s}$ as

$$* :: \Re^{+s} \times \Re^{+s} \to \Re^{+s}, \quad /_l :: \Re^{+s} \times \overline{\Re^{+s}} \to \Re^{+s}, \quad /_r :: \Re^{+s} \times \overline{\Re^{+s}} \to \Re^{+s},$$

and $/_l$, $/_r$ in $\Re^{+}$ and all the operators in B are defined as

$$\begin{aligned} a/_l b &= a/b && \text{if} \quad b \neq 0.0,\\ a/_l b &= 0.0 && \text{if} \quad b = 0.0,\\ a/_r b &= a/b && \text{if} \quad b \neq 0.0,\\ a/_r b &= \top_{\Re} && \text{if} \quad b = 0.0,\\ b_1 \circ_B b_2 &= \min_{\mathrm{B}}(b_1, b_2) && \text{for} \quad \circ \in \{*, /_l, /_r\}. \end{aligned}$$

Observe that, as $b \neq 0.0$ is weaker than the condition b is ground, there is more pruning than that obtained by delaying the evaluation of the constraint until it is linear.

Notice that this same proposal can be used to specify the rounding mode of floating point computations. For instance, a constraint such as '$x^2 = z$'/2 can be defined as

$$\begin{aligned} x^2 = z \Leftrightarrow x &\sqsubseteq \langle \overline{\mathrm{nearest}_d}(\overline{\mathrm{sqrt}}(\min(z))), \mathrm{nearest}_e(\mathrm{sqrt}(\max(x)))\rangle,\\ z &\sqsubseteq \langle \overline{\mathrm{nearest}_d}(\min(x)\overline{*}\min(x)), \mathrm{nearest}_e(\max(x) * \max(x))\rangle, \end{aligned}$$

where sqrt, $\mathrm{nearest}_d$, and $\mathrm{nearest}_e$ are declared on $\Re^{+s}$ as

$$\mathrm{sqrt} :: \Re^{+s} \to \Re^{+s}, \quad \mathrm{nearest}_d :: \Re^{+s} \to \Re^{+s}, \quad \mathrm{nearest}_e :: \Re^{+s} \to \Re^{+s}$$

and are defined in B as the identity and, for any $x \in \Re^{+}$,

$\mathrm{sqrt}(x)$	returns the square root of x,
$\mathrm{nearest}_d(x)$	returns the nearest floating point number $\leq x$,
$\mathrm{nearest}_e(x)$	returns the nearest floating point number $\geq x$.

This constraint will lead to better pruning than using the constraint '$x * x = z$'. Note that the proposal in Benhamou et al. [1999] which combines the evaluation of primitive constraints with specific methods for solving nonlinear constraints is likely to be more efficient although less declarative.

8.3 The *precision*/1 Map As a Normalization Rule

When there exist more than one solution, some sort of domain splitting should be applied in order to look for solutions in each of the resulting partitions of the problem. When the constraint system supports multiple domains, the precision map (see Definition 14) provides a useful way to normalize the heuristics for value ordering.

Example 16. The well-known *first fail principle* usually chooses the variable constrained by the smallest range. However, in systems supporting multiple domains, it is not always clear which is smallest. One way to compare the ranges is to use the map *precision*/1 defined for each computation domain. To see this, consider a set of variables $X = \{x_1, \ldots, x_n\}$ and constraint store

$S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X$ where for each $i \in \{1, \ldots, n\}$, c_i is the simple interval constraint in S with constrained variable x_i. Suppose that S is *divisible* (i.e., S can be partitioned into, at least, two consistent stores). Then the first fail principle can be emulated with the procedure $choose_{FirstFail}$ which chooses the "smallest" constraint in S that is *divisible* (i.e., one whose range can be partitioned into two consistent parts).[8]

P*recondition:* $\{S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X$ is divisible$\}$

$choose_{FirstFail}(S) = c_j;$

P*ostcondition:* $\{j \in \{1, \ldots, n\}, c_j$ is divisible and

$\forall i \in \{1, \ldots, n\} \backslash \{j\} : c_i$ divisible $\Longrightarrow precision_{L_j}(c_j) \preceq precision_{L_i}(c_i)\}$.

8.4 Disjunctive Constraints

It is well known that disjunctive constraints are sometimes useful for formulating a solution to a problem. In logic programming, one easy but inefficient way to handle disjunctive constraints is to create choice points. A more efficient approach that avoids the use of choice points would be to generalize the solution proposed by the clp(FD) system in Codognet and Diaz [1996a]. This is defined for the class of disjunctive constraints of the form $c_1 \vee \cdots \vee c_n$ where each c_i $(1 \leq i \leq n)$ has the form $x_1 \sqsubseteq r_1^i \wedge \cdots \wedge x_k \sqsubseteq r_k^i$ and is constrained on the same variables, that is, $\{x_1, \ldots, x_k\}$. The importance of this kind of disjunction is that, as observed in Codognet and Diaz [1996a], "nearly all current uses of constructive disjunction fit in this case." We can generalize this solution by incorporating, in the definition of the constraint $x \sqsubseteq r$, range union operations[9] as $x \sqsubseteq r_1 \vee \cdots \vee r_n$. This delays the creation of choice points and can lead to a reduction in the size of the search tree.

An alternative technique, adapted from an idea shown in Van Hentenryck et al. [1998], would be to define the constructive disjunction, that is, to consider the range as an interval and the lub as defined for interval lattices [Slavík 1986]. Thus the lub of a set of ranges would be the range whose lower bound was the glb of the lower bounds and the upper bound, the lub of the upper bounds. However, this can lead to the addition of infeasible values. To see this, consider the domain *Integer* and let $r_1 = \langle 1_], 4_] \rangle$ and $r_2 = \langle 6_], 10_] \rangle$. Then the range, $\text{lub}_{\mathcal{C}^{\{v\}}_{Integer}}(v \sqsubseteq r_1, v \sqsubseteq r_2) = v \sqsubseteq \langle 1_], 10_] \rangle$, includes the value 5 which was not in either r_1 or r_2.

8.5 Global Constraints

Recently Hickey [2000] has shown that contraction algorithms (such as the Taylor contractor) can be implemented in a declarative style in the CLIP system. CLIP is a CLP(Interval($\Re$)) system in which constraints are decomposed into sets of primitive constraints that are sent to a constraint-solving engine providing support for interval arithmetic and where computation is done over

[8]It is straightforward to include more conditions, for example, if c_i, c_k, c_j have the same (minimum) precision, the "left-most" domain can be chosen, that is, $c_{minimum(i,k,j)}$.

[9]In fact, $\vee$ is not the union operator but a composition operator since our ranges do not correspond to sets but to elements in our interval lattices.

the floating point intervals associated to the constrained variables. We have already shown in Section 8.2, how our solver can handle floating point computations and have shown how the (relational) product operator '$x * y = z$'/3 for the floating point domain could be defined. In the same style, it is straightforward to define the relational floating point version of all the classical real interval operators, that is, $*$, $+$, $-$, and / [Moore 1966]. For example, the definition of the `plus` constraint in Example 13 corresponds to the classical definition of the $+$ operator for interval arithmetic. It is straightforward to adapt this definition to the floating point real domain by using the operators nearest_d and nearest_e as done for the constraint '$x * y = z$'. Also, throughout the paper we have shown that the declarative nature of our solver allows the user to define (possibly cooperative, generic and/or overloaded) high-level constraints. As CLP(Interval($\Re$)) is an instance of our framework, it is natural to expect that the CLIP approach for generating global contractors can also be adapted for our solver.

Note that, for the system domains (i.e., real, FD, set, or Boolean domains), the implementation can provide specialized low-level optimizations. For example, for the real domain, well-known implementations of functions such as exp/1 or cos/1 can be used. Observe that these are now provided in our prototype implementation for $clp(\mathcal{L})$ since these are already available in SICStus, the system in which our prototype is constructed.

9. RELATED WORK

Indexical-based implementations. The indexical approach from which our framework is derived was first implemented by Codognet and Diaz [1996a], where finite interval constraints of the form $x\ in\ r$ were efficiently implemented using an extension of the WAM [Diaz and Codognet 1993]. Well-known CLP systems such as SICStus [Sicstus Manual 1994] and IF/Prolog [If/Prolog 1994] now integrate the $x\ in\ r$ constraint to provide a glass box solver for FD, the finite domain of integers.

In Codognet and Diaz [1993], the idea was extended to the clp(FD/B) system that integrates a Boolean solver into the existing FD solver. A version of the clp(FD/B), called clp(B), developed by Codognet and Diaz [1994] solely for the Boolean domain, has an efficiency that was, on average, an order of magnitude faster than most of the existing Boolean solvers including, surprisingly, some special-purpose Boolean solvers.

More recently, Georget and Codognet [1998] used the indexical approach to implement a generic language for semiring-based constraint satisfaction (also on FD) and demonstrated its efficiency with respect to dedicated systems. Moreover, Goualard et al. [1999] described a system called *DecLic* that extends the clp(FD) solver to provide an efficient constraint solver for continuous domains (i.e., the real domain).

These indexical-based systems therefore show that the indexical approach can obtain competitive efficiency for the Boolean, finite, and continuous domains. Moreover, these systems demonstrate that the implementation of high-level constraints (e.g., interval arithmetic) as rules (e.g., combination of simple primitive constraints) is fairly efficient. We therefore anticipate that we can adapt the techniques used for the implementation of clp(FD), clp(B), and *DecLic* to our constraint system. Despite the expected loss of some optimizations for specific domains due to the generality of our framework, we expect to obtain reasonably competitive performance compared to domain-specific systems.

Other optimizations in the low level can be incorporated as done in the clp(FD) system. For example it is straightforward to incorporate the indexical dom(y) to return the whole range associated to y. In this case a constraint such as $x \sqsubseteq \text{dom}(y)$ is equivalent to the constraint $x \sqsubseteq \langle \min(y), \max(y) \rangle$.

Interval reasoning. Interval arithmetic, on which the indexical approach to constraint solving is based, has been applied to constraint satisfaction problems over numeric domains [Benhamou 1995; Lee and van Emden 1993; Older 1989; Benhamou and Older 1997] and, in particular, to floating point numbers on relational programming [Cleary 1987]. In this latter application, interval computations are used to approximate a computed real number. This concept of approximation, which works well on numeric domains, does not generalize since it assumes that the closest value smaller (respectively higher) than any computed value is computable. To ensure that this property holds in our more generic framework, we have defined a new system of approximation that is applicable to any (possibly infinite) lattice.

Older and Vellino [1993] presented a lattice-theoretic semantics for numeric interval constraints that aims to capture the properties of both the primitive interval operations and the constraint propagation networks created from them. Based on lattice theory, some analogies with respect to our proposal can be detected: (1) the computation domain has a lattice structure and is constructed from the bounds of the intervals; (2) the theory can be applied with infinite precision (i.e., without approximating a real to a floating point number) although only on reals; (3) the operators are assumed to maintain properties over the computation domain that are also maintained by our constraint operators (e.g., monotonicity); (4) the propagation process is based on a fixed-point semantics. In spite of the similarities, there are a number of aspects that made this approach very different from our proposal: (1) the framework is developed exclusively for numeric domains; (2) we provide a control mechanism, at the user level, for the propagation by allowing the constraint operators to be defined directly on the bounds of the interval; in this sense, Older and Vellino [1993] did not treat the issue of the transparency of their theory; (3) the theory proposed in Older and Vellino [1993] is "quite abstract and therefore somewhat remote from actual implementations," whereas our theory can be directly implemented; (4) we treat the termination issue even for nonnumeric domains; (5) solver cooperation was not treated at all in Older and Vellino [1993] and (6) implementation issues were neglected.

Generic constraint solving procedures. Apt [1999, 2000] proposed a framework for constraint propagation based on chaotic iteration algorithms for partially ordered domains. A key observation in these papers was that most constraint propagation algorithms presented in the literature can be expressed as direct instances of these algorithms. There are many similarities between the frameworks described here and in Apt [1999] and these, together with the main differences, were discussed in detail in Fernández and Hill [1999a], where we showed that the process of constraint propagation in our operational schema can be viewed as a process of function evaluation in the chaotic iteration algorithm given in Apt [1999]. One difference is that, in Apt [1999], the chaotic iteration approach was specialized for a constraint satisfaction problem where the set of constraints to be solved was interpreted as sets of possible solutions whereas a set of interval constraints in our framework embodies more: the intended interaction between the variables in the constraint propagation. A second important difference is that the chaotic iteration approach assumes the finite chain property[10] whereas our domains do not necessarily possess this property. Further study on how the idea of computing an approximate solution via a precision map as defined in this paper can be adapted for a chaotic iteration algorithm is needed. Note that a technique such as this could be useful in extending the framework described in Apt [2000] to domains not satisfying the finite chain property.

General frameworks for solving soft constraints (i.e., constraints with an associated confidence value, for example, cost, uncertainty, or degree) have been described by both Schiex et al. [1995] and Bistarelli et al. [1995]. The framework in Schiex et al. [1995] was defined for any finite totally ordered domain which has a specific operation satisfying certain properties whereas the framework in Bistarelli et al. [1995], was defined for a finite semiring structure. In neither case was the domain allowed to be infinite.

A generic form of constraint propagation called *generalized propagation (GP)*, proposed in [Le Provost and Wallace 1993], is applicable to arbitrary computation domains. Note that, unlike our proposal, the constraints are not defined generically but use any available constraint over any computation domain to express restrictions on problem variables. One drawback of GP compared to our approach, is that termination of the search for answers to a propagation constraint is not guaranteed and the entire responsibility for ensuring termination remains with the programmer.

Solver communication and cooperation. Baader and Schulz [1995] provided an abstract framework for combining different and (unlike in our proposal) independently defined constraint languages and constraint solvers. Thus, they were primarily concerned with the properties that such a combined solution structure should satisfy.

A general scheme for solver cooperation was proposed by Hofstedt [2000]. In this paper, domains were defined by using "Σ-Structures" in a sorted language and a constraint was a relation over an n-ary Cartesian product of the domains. As for our framework, solvers are combined by means of the Cartesian product of the domains. However, Hofstedt [2000] assumed that each component domain has its own associated solver built into the system and therefore focused on the interface between the component solvers; the complete system consisted of the

[10]A domain L satisfies the finite chain property if every increasing sequence $a_0 \leq a_1 \leq a_2 \dots$ of its elements eventually stabilises, that is, for some $j > 0$, $a_i = a$ for $i \geq j$ and $a \in L$.

interface plus a set of built-in constraint systems. In contrast, in our proposal the high-level constraints determine the possible cooperation that can occur between the domains and their solvers and these constraints may be defined by the user or system. Note that the flexibility of these high-level constraints implies that the solver interface defined by Hofstedt [2000] could be implemented in our system.

Constraint systems such as CLP(BNR) [Benhamou and Older 1997] and Prolog IV [N'Dong 1997] provide some support for cooperation between solvers. However, in these languages, the solver cooperation is mainly limited to Booleans, reals, naturals lists, and trees. Moreover, this cooperation is usually hard-wired and built into the language.

Interval lattice theory. The interval topology which is the lattice of closed intervals of a lattice [Birkhoff 1967] appears to be similar to the intervals on which our constraints are based and it would have been useful if we could have based our interval constraints on this formalism. However, in our framework, to allow for approximations in the continuous domains, the intervals are not necessarily closed, so that the ranges for a domain $L \in \mathcal{L}$ are not necessarily meet- or join-complete sublattices of L and do not form an interval topology. Second, in order to guarantee the monotonicity of our interval constraints (using the indexical functions $\min/1$, $\overline{\mathrm{val}}/1$ and $\max/1$, $\mathrm{val}/1$) and identify, prior to the resolution step, those constraints that do not lead to further propagation, we needed to distinguish between a domain and its mirror.

10. CONCLUSIONS

We have defined a framework for constraint solving over lattices and illustrated with many examples the versatility and expressivity of this approach. For maximum generality and to allow for any lattice, finite or infinite, discrete or continuous, we have constructed the interval domains in several stages, each stage taking advantage of the lattice structures inherited from the underlying computation domains on which the interval domains are built. Thus, we first defined and added the bracket domain B to each computation domain L to create the right bounded domain L^s for open and closed (right) bounds for the intervals. We then defined the symmetric mirror domain $\overline{L^s}$ so as to allow for the left bounds of intervals. These bounds were then combined using the direct product of lattices to form the range elements of the interval domain R^s_L. Finally, we added the variable to be constrained to the given range to form the interval constraint $x \sqsubseteq r$.

When defining the elements of the bounded computation domain L^b, we introduced two additional constructs. One, which generalizes an idea from Codognet and Diaz [1996a], was indexicals $\max(x)$, $\mathrm{val}(x)$, for the right bounded domain and $\min(x)$, $\overline{\mathrm{val}(x)}$, for the left bounded domain. These provide necessary links between the ranges for the constrained variables and give the user transparent control over the constraint propagation. The other was that of an operator $\circ_L$ which maps a domain constructed from several, possibly distinct, computation domains $L_1, \ldots, L_n$ to another, possibly different, codomain L. This, combined with the indexicals, allows a one-way communication from the domains

$L_1^b, \ldots, L_n^b$ to the domain L^b. Finally, for full solver cooperation, we have shown how a high-level constraint defined as a relation over a domain constructed from a set of computation domains can provide unrestricted communication between these domains. Notice that this formalization of the framework is new even when restricted to just the finite domains of integers.

We have presented an operational schema for solving these constraints and proved it correct. In the case of the nonfinite domains, termination of the procedure can only be guaranteed by letting the solver return an approximation to the correct result. An idea from Sidebottom and Havens [1992] for controlling accuracy in the processing of disjoint intervals over the reals was adapted for our lattice domains. The special operator $precision_L/1$ that maps the domain elements to nonnegative reals $\Re^+$ and a limit element $\varepsilon \in \Re^+$ that controls the degree of the approximation were introduced. Observe that the notion of our precision operator corresponds, in some sense, to a change of domain where the domains (i.e., intervals) which are "too small" but still consistent are not considered in the lattice. The basic operational schema was then adapted so as to check, using these precision and limit constructs, for just an approximation to the fix-point. With this modification of the schema, we proved that such a procedure terminates with an approximate solution.

Observe that the framework, being applicable to any lattice, provides support for all the existing practical domains in CLP (e.g., reals, integers, sets, and Booleans). Moreover, by using lattice combinators, new compound domains and their solvers can easily be obtained from previously defined domains such as these. We have imposed the restriction that the sets of constrained variables associated to each computation must be disjoint. However, it should be possible to remove such a restriction and consider sublattices of lattices as computation domains. These could provide a means of having variable sets of a sublattice being also allowed as variables in the main lattice. This is another topic for future work.

To demonstrate our framework is realizable in a practical setting, we have developed the CLP language $clp(\mathcal{L})$ and built a prototype implementation using CHR's [Frühwirth 1998]. This prototype supports a set of built-in domains as well as user-defined domains. Note that all the examples in this paper have been solved with this implementation of $clp(\mathcal{L})$. We note that this implementation is only a prototype and it does not compete with the CHRs although it may be considered as a CHR module. In fact, this system shows the feasibility of our ideas and was not designed with efficiency in mind. Thus the development of an efficient implementation is future work.

APPENDIX A. PROOFS

Remark 1. As in the main part of the paper, $L \in \mathcal{L}$, V_L is the set of variables associated with L, $\mathcal{V}_{\mathcal{L}} = \cup\{V_L | L \in \mathcal{L}\}$, R_L^b is the interval domain over L, $X \in \wp_f(\mathcal{V}_{\mathcal{L}})$ is the set of constrained variables, $\mathcal{C}^X$ is the set of all interval constraints for X, and $\mathcal{SS}^X$ is the set of all simple stable constraint stores for X. In the proofs, elements of the bounded computation domain are denoted as a_b or (a, b), depending on context.

PROPOSITION 3. *Let $L' \in \{L^s, \overline{L^s}\}$ and $t \in L'$. Then,*

$$(1)\ \overline{\overline{L'}} = L', \quad (2)\ \overline{\overline{t}} = t.$$

PROOF. We prove the cases separately.

(1) Observe that

$$\overline{\overline{L^s}} =^1 \overline{\overline{(L, B)}} =^1 (\hat{\hat{L}}, B) =^2 (L, B) =^1 L^s, \tag{4}$$

where equality, $=^1$ follows from Definition 2, and $=^2$ follows from the duality definition for lattices. Thus, if $L' = L^s$, then the result follows. Moreover, if $L' = \overline{L^s}$, then $\overline{\overline{L'}} = \overline{\overline{\overline{L^s}}} = \overline{L^s} = L'$.

(2) Observe that, for some $a \in L \cup \hat{L}$ and $t = a_b \in \{L^s, \overline{L^s}\}$, we have

$$\overline{\overline{t}} = \overline{\overline{a_b}} =^1 \overline{(\hat{a}, b)} =^1 (\hat{\hat{a}}, b) =^2 (a, b) =^1 a_b = t,$$

where equality $=^1$ follows from Definition 2 and $=^2$ from the duality definition for lattices. □

PROPOSITION 4. *Suppose $\circ$ is a constraint operator for L^s. Then,*

$$\circ \textit{ is monotonic;} \tag{a}$$

$$\overline{\circ} \textit{ is a constraint operator (for } \overline{L^s}\textit{);} \tag{b}$$

$$\circ \textit{ is the mirror of } \overline{\circ} \textit{ i.e., } \circ \equiv \overline{\overline{\circ}}. \tag{c}$$

PROOF. Suppose that $\circ :: L^s_1 \times \cdots \times L^s_n \to L^s$ is a constraint operator. We prove the cases separately.

(a) Suppose $t_i \preceq t'_i$ for $i \in \{1, \dots, n\}$, where $t_i = (a_i, b_i)$ and $t'_i = (a'_i, b'_i)$. We need to show

$$\circ(t_1, \dots, t_n) \preceq \circ(t'_1, \dots, t'_n). \tag{5}$$

Observe that

by the product of lattices: $a_i \preceq a'_i$; moreover if $a_i = a'_i$ then $b_i \preceq b'_i$;

by monotonicity of $\circ_L$: $\circ_L(a_1, \dots, a_n) \preceq \circ_L(a'_1, \dots, a'_n)$.

If $\circ_L(a_1, \dots, a_n) \prec \circ_L(a'_1, \dots, a'_n)$ then (5) holds by the product of lattices and Definition 3.

Otherwise, $\circ_L(a_1, \dots, a_n) = \circ_L(a'_1, \dots, a'_n)$. There are two cases:

(1) $\forall i \in \{1, \dots, n\}, a_i = a'_i$. Then, by monotonicity of $\circ_B$,

$$\circ_B(b_1, \dots, b_n) \preceq \circ_B(b'_1, \dots, b'_n).$$

(2) $\exists i \in \{1, \dots, n\}, a_i \prec a'_i$. Then $\circ_L$ is not a strict monotonic function and, by Definition 3, $\circ_B$ is a constant, that is, $\circ_B(b_1, \dots, b_n) = \circ_B(b'_1, \dots, b'_n)$.

Thus, in both cases, (5) holds by the product of lattices and Definition 3.

(b) Observe that

$$\overline{\circ} :: \overline{L^s_1} \times \cdots \times \overline{L^s_n} \to \overline{L^s} \qquad \text{(by Definition 3)},$$

$$\overline{\circ} :: \widehat{L_1}^s \times \cdots \times \widehat{L_n}^s \to \hat{L}^s \qquad \text{(by Definition 2)},$$

where, for $i \in \{1, \dots, n\}$, $\hat{L}_i \in \mathcal{L} \cup \hat{\mathcal{L}}$ and $\hat{L} \in \hat{\mathcal{L}}$.

Assuming the notation of Definition 3, we have, if $t_i = (a_i, b_i)$ $(1 \le i \le n)$, then $\circ(t_1, \dots, t_n) = \circ_L(a_1, \dots, a_n)_{\circ_B(b_1,\dots,b_n)}$.

Let $\circ_{\hat{L}}$ :: $\widehat{L_1} \times \cdots \times \widehat{L_n} \to \hat{L}$ be the dual operator to $\circ_L$ so that, if $\circ_L(a_1, \dots, a_n) = a$, then $\circ_{\hat{L}}(\hat{a}_1, \dots, \hat{a}_n) = \hat{a}$. Then, by the duality principle of lattices, $\circ_{\hat{L}}$ is monotonic in $\hat{L}$. Moreover, $\circ_L$ is strict monotonic whenever $\circ_{\hat{L}}$ is. Hence, we have

$$
\begin{aligned}
\overline{\circ}(\overline{t_1}, \dots, \overline{t_n}) &= \overline{\circ(t_1, \dots, t_n)} && \text{(by Definition 3)}\\
&= \overline{\circ_L(a_1, \dots, a_n)_{\circ_B(b_1,\dots,b_n)}} && \text{(by Definition 3)}\\
&= (\widehat{\circ_L(a_1, \dots, a_n)}, \circ_B(b_1, \dots, b_n)) && \text{(by Definition 2)}\\
&= (\circ_{\hat{L}}(\widehat{a_1}, \dots, \widehat{a_n}), \circ_B(b_1, \dots, b_n)) && \text{(by the definition of } \circ_{\hat{L}})\\
&= \circ_{\hat{L}}(\widehat{a_1}, \dots, \widehat{a_n})_{\circ_B(b_1,\dots,b_n)} && \text{(by Definition 3).}
\end{aligned}
$$

Thus $\overline{\circ}$ is a constraint operator (for $\overline{L^s}$) as defined in Definition 3.

(c) Observe that

$$
\begin{aligned}
\overline{\circ} &:: \overline{L^s_1} \times \cdots \times \overline{L^s_n} \to \overline{L^s} && \text{(by Definition 3);}\\
\overline{\overline{\circ}} &:: \overline{\overline{L^s_1}} \times \cdots \times \overline{\overline{L^s_n}} \to \overline{\overline{L^s}} && \text{(by Definition 3);}\\
\overline{\overline{\circ}} &:: L^s_1 \times \cdots \times L^s_n \to L^s && \text{(by Proposition 3(1)).}
\end{aligned}
$$

Also if $t_i \in L^s_i$ for all $i \in \{1, \dots, n\}$,

$$
\begin{aligned}
\overline{\overline{\circ}}(t_1, \dots, t_n) &= \overline{\overline{\circ}}(\overline{\overline{t_1}}, \dots, \overline{\overline{t_n}}) && \text{(by Proposition 3(2))}\\
&= \overline{\overline{\circ(t_1, \dots, t_n)}} && \text{(by Definition 3, applied twice)}\\
&= \circ(t_1, \dots, t_n) && \text{(by Proposition 3(2)).}
\end{aligned}
$$

Therefore $\overline{\overline{\circ}}$ is equivalent to $\circ$. □

Proposition 1. (*See Section* 3.4.)

Proof. Suppose that

$$
\begin{aligned}
r &= \langle \overline{s}, t \rangle, & r' &= \langle \overline{s'}, t' \rangle,\\
s &= (a, b_1), & s' &= (a', b'_1),\\
t &= (c, b_2), & t' &= (c', b'_2).
\end{aligned}
$$

By hypothesis, $r \preceq r'$ and, by Definition 5 and by the product of lattices (i.e., direct product),

$$\overline{s} \preceq \overline{s'}; \tag{6}$$

$$t \preceq t'. \tag{7}$$

From (6):

$$
\begin{aligned}
\overline{s} \preceq \overline{s'} &\Rightarrow^1 (\hat{a}, b_1) \preceq (\hat{a'}, b'_1)\\
&\Rightarrow^2 \hat{a} \prec \hat{a'} \text{ or } \hat{a} = \hat{a'} \text{ and } b_1 \preceq b'_1\\
&\Rightarrow^3 a' \prec a \text{ or } a = a' \text{ and } \}_1 \preceq \}'_1\\
&\Rightarrow^2 \begin{cases} \text{if } a \prec a' \text{ then } s' \prec s\\ \text{if } a = a' \text{ then } s \preceq s', \end{cases}
\end{aligned} \tag{8}
$$

where $\Rightarrow^1$ follows from Definition 2, where $\Rightarrow^2$ follows from the product of lattices (i.e., the lexicographic product) and $\Rightarrow^3$ follows from the duality principle for lattices in Section 2.

We suppose that r' is inconsistent. Then, by Definition 6, we have three cases:

$$t' \prec s'; \tag{i}$$
$$s' \not\preceq t'; \tag{ii}$$
$$s' = a')\ \text{and}\ t' =' a\}'_2. \tag{iii}$$

In the following:

$\Rightarrow^4$ follows from Equation (8);
$\Rightarrow^5$ follows from (7);
$\Rightarrow^6$ follows from Definition 6;
$\Rightarrow^7$ follows from Definition 2;
$\Rightarrow^8$ follows from the product of lattices (i.e., the lexicographic product);
$\Rightarrow^9$ follows from a contradiction;
$\Rightarrow^{10}$ follows from Case (i) (i.e., $t' \prec s'$);
$\Rightarrow^{11}$ follows from (7) since $t \preceq t'$ and, by the product of lattices (i.e., lexicographic product), $c \preceq c'$;
$\Rightarrow^{12}$ follows from Case (ii) so that $s' \not\preceq t'$. Then, by Definition 2 and by the product of lattices (i.e., lexicographic product), $a' \not\preceq_L c'$ and thus $a \not\preceq_L c'$.

As shown in (8), $a \prec a'$ or $a = a'$. Suppose first $a \prec a'$. Then,

$$\begin{aligned}
&\text{if (i)} \Rightarrow^5 t \prec s' \Rightarrow^4 t \prec s \Rightarrow^6 r \text{ is inconsistent;}\\
&\text{else if (ii)} \Rightarrow^4 s \not\preceq t' \Rightarrow^5 s \not\preceq t \Rightarrow^6 r \text{ is inconsistent;}\\
&\text{otherwise if (iii)} \Rightarrow^8 s \prec s' \Rightarrow^4 \mathit{false}.
\end{aligned}$$

Suppose now $a = a'$. Then,

$$\text{if (i)} \Rightarrow^4 s \preceq s' \Rightarrow \begin{cases} s = s' \Rightarrow^{10} t' \prec s \Rightarrow^5 t \prec s \Rightarrow^6 r \text{ is inconsistent;}\\ s \prec s' \Rightarrow^7 s = a) \text{ and } s' = a] \Rightarrow^{10} t' \prec a] \\ \qquad \Rightarrow^8 t' \preceq a) \Rightarrow^5 t \preceq a) \Rightarrow^6 r \text{ is inconsistent;}\end{cases}$$

else if (ii) $\Rightarrow^4\ s \not\preceq t' \Rightarrow^5 s \not\preceq t \Rightarrow^6 r$ is inconsistent;

$$\text{otherwise if (iii)} \Rightarrow^4 s = a') \Rightarrow^5, \begin{cases} t = t' \Rightarrow t = (a', b'_2) \Rightarrow^6 r \text{ is inconsistent;}\\ t \prec t' \begin{cases} \Rightarrow^8 c = a' \text{ and } b_2 \preceq b'_2 \Rightarrow t = a'\}_2 \\ \qquad \Rightarrow^6 r \text{ is inconsistent;}\\ \Rightarrow^8 c \prec a' \Rightarrow^8 t \prec s \Rightarrow^6 r \text{ is inconsistent.}\end{cases}\end{cases}$$

Thus, in all cases, r is inconsistent. □

PROPOSITION 5. *Let $S, S' \in \mathcal{SS}^X$ and let also $X' \subseteq X$ and $C \in \mathcal{S}^{X'}$. Then, if $S \cup C \mapsto S'$, $S' \preceq S$.*

PROOF. Suppose for each $x \in X$, x is constrained by the constraints $c_x \in S$ and $c'_x \in S'$. Also, if $x \in X'$, suppose that C_x is the set of constraints in C with constrained variable x.

Then, by Definition 12,

$$\begin{array}{ll} c'_x = \cap_L(C_x \cup \{c_x\}) & \text{if } x \in X', \\ c'_x = c_x & \text{otherwise.} \end{array}$$

Therefore, by Definition 8 and the resulting contractance property, $c'_x \preceq c_x$ for each $x \in X$. As consequence, by Definition 9, $S' \preceq S$. □

PROPOSITION 6. *Suppose $S, S' \in \mathcal{SS}^X$ where $S \preceq S'$. Then, if S' is inconsistent, S is also inconsistent.*

PROOF. Suppose that S' is inconsistent. Then, by Definition 9, there is, at least, one inconsistent constraint $c'_x = x \sqsubseteq r' \in S'$ (for some $x \in X$). By Definition 7, this means that r' is inconsistent.

Let $c_x = x \sqsubseteq r$ be the constraint for x in S. By hypothesis, $S \preceq S'$ so that by Definition 9, $c_x \preceq c'_x$ for all $x \in X$, and by Definition 7, $r \preceq r'$. Thus, by Proposition 1, r is also inconsistent. Thus, by Definition 7, c_x is also inconsistent and hence, by Definition 9, S is inconsistent. □

LEMMA 1. *Suppose that $c \preceq c'$ are simple consistent constraints for $L \in \mathcal{L}$ constraining the same variable $y \in X$ and suppose also that $c' = y \sqsubseteq \langle \overline{t'}, t' \rangle$ for some $t' \in L^s$. Then $c = c'$.*

PROOF. Suppose that $t' = a_b$, for some $a \in L$. Then, as c' is consistent, by Definition 6, $t' \neq a_)$ so that $t' = a_]$. Suppose also that $c = y \sqsubseteq \langle \overline{s}, t \rangle$. Then, by Definition 7, $\langle \overline{s}, t \rangle \preceq \langle \overline{a_]}, a_] \rangle$, and, by Definition 5,

$$\overline{s} \preceq \overline{a_]} \text{ and } t \preceq a_].$$

Then, by Definition 2 and by the product of lattices (i.e., the lexicographic product),

$$(\overline{s} = \overline{a_]} \text{ or } \overline{s} = \overline{a_)} \text{ or } \overline{s} = \overline{a_{1_{b'}}} \text{ and } \widehat{a_1} \prec \hat{a})$$

and

$$(t = a_] \text{ or } t = a_) \text{ or } t = a_{2_{b'}} \text{ and } a_2 \prec a).$$

By the duality principle of lattices in Section 2, this is equivalent to

$$(s = a_] \text{ or } s = a_) \text{ or } s = a_{1_{b'}} \text{ and } a \prec a_1)$$

and

$$(t = a_] \text{ or } t = a_) \text{ or } t = a_{2_{b'}} \text{ and } a_2 \prec a.).$$

However, c is consistent so that, by Definition 7, $\langle \overline{s}, t \rangle$ is consistent. By Definition 6, this means that $s \preceq t$ and, if $s = a_)$ then $t \neq a_]$. The only case for which this holds is when $s = t = a_]$. □

LEMMA 2. *Let $S_1, S_2 \in \mathcal{SS}^X$ be two consistent stores such that $S_1 \preceq S_2$ and $c, c_2 \in \mathcal{C}^X$ such that $c \leadsto^{S_2} c_2$. Then, there exists $c_1 \in \mathcal{C}^X$ such that $c \leadsto^{S_1} c_1$ and $c_1 \preceq c_2$.*

PROOF. Let $c = x \sqsubseteq \langle \overline{s}, t\rangle$ where $x \in X$ and $x \in V_L$ for some $L \in \mathcal{L}$. Then as $c \leadsto^{S_2} c_2$, by Definition 11, $c_2 = \text{eval}(S_2, c)$ and c_2 is simple. Then it follows from the Definition 10 and Definition 7 that

$$\begin{gathered} c_2 = x \sqsubseteq \langle \text{eval}(S_2, \overline{s}), \text{eval}(S_2, t)\rangle, \\ \text{eval}(S_2, \overline{s}) \in \overline{L^s} \ \text{ and } \ \text{eval}(S_2, t) \in L^s. \end{gathered} \tag{9}$$

Suppose that $c_1 = x \sqsubseteq \text{eval}(S_1, c)$. Then, again, it follows from Definition 10 that

$$c_1 = x \sqsubseteq \langle \text{eval}(S_1, \overline{s}), \text{eval}(S_1, t)\rangle.$$

We have to prove that $c \leadsto^{S_1} c_1$ and $c_1 \preceq c_2$ which means that, by Definition 7 and Definition 11, we have to show that c_1 is simple and that

$$\langle \text{eval}(S_1, \overline{s}), \text{eval}(S_1, t)\rangle \preceq \langle \text{eval}(S_2, \overline{s}), \text{eval}(S_2, t)\rangle. \tag{10}$$

However, by Definition 5, if relation (10) holds, c_1 is simple. Thus, by the product of lattices (i.e., direct product), we just have to show that

$$\text{eval}(S_1, \overline{s}) \preceq \text{eval}(S_2, \overline{s}) \text{ and} \tag{i}$$
$$\text{eval}(S_1, t) \preceq \text{eval}(S_2, t). \tag{ii}$$

Let $n(term)$ be the number of operators in $term$. We prove (i) by induction on $n(\overline{s})$. The proof of (ii) is similar and omitted.

—*Base case:* $n(\overline{s}) = 0$. If $\overline{s} \in \overline{L^s}$, then, by Definition 10, $\text{eval}(S_1, \overline{s}) = \text{eval}(S_2, \overline{s}) = \overline{s}$. If $\overline{s} \notin \overline{L^s}$, then $\overline{s} = \min(y)$ or $\overline{s} = \text{val}(y)$ for some $y \in X$. Thus there exists $c_y = y \sqsubseteq \langle \overline{s_y}, t_y\rangle \in S_1$ and $c'_y = y \sqsubseteq \langle \overline{s'_y}, t'_y\rangle \in S_2$ so that as, by hypothesis $S_1 \preceq S_2$, we have

$$S_1 \preceq S_2 \Rightarrow^1 c_y \preceq c'_y \Rightarrow^2 \langle \overline{s_y}, t_y\rangle \preceq \langle \overline{s'_y}, t'_y\rangle \Rightarrow^3 \overline{s_y} \preceq \overline{s'_y} \quad \text{and} \quad t_y \preceq t'_y, \tag{11}$$

where $\Rightarrow^1$ follows from Definition 9, $\Rightarrow^2$ from Definition 7, and $\Rightarrow^3$ from the product of lattices (i.e., direct product) and Definition 5.

Suppose first that $\overline{s} = \min(y)$. Then, by Definition 10,

$$\text{eval}(S_1, \overline{s}) = \overline{s_y} \text{ and } \text{eval}(S_2, \overline{s}) = \overline{s'_y}.$$

Therefore, by (11), $\text{eval}(S_1, \overline{s}) \preceq \text{eval}(S_2, \overline{s})$.

Second, suppose that $\overline{s} = \text{val}(y)$. By (9) $\text{eval}(S_2, \overline{s}) \in \overline{L^s}$ and by Definition 10, $\text{eval}(S_2, \overline{s}) = \overline{s'_y}$ and $s'_y = t'_y$. Therefore, as S_1 and hence c_y are consistent, it follows from (11) and Lemma 1 that $s_y = t_y = s'_y$. Thus, by Definition 10, $\text{eval}(S_1, \overline{s}) = \overline{s_y}$ so that $\text{eval}(S_1, \overline{s}) = \text{eval}(S_2, \overline{s})$.

—*Nonbase case:* $n(\overline{s}) > 0$. Suppose $\circ :: L_1^s \times \cdots \times L_n^s \to L^s$ is a constraint operator (for L^s). Then, by Proposition 4(b), $\overline{\circ}$ is also a constraint operator (for $\overline{L^s}$). Then,

$$n(\overline{s}) > 0 \Rightarrow \overline{s} = \overline{\circ(s_1, \ldots, s_n)} =^1 \overline{\circ}(\overline{s_1}, \ldots, \overline{s_n})$$
$$\Rightarrow^4 \begin{cases} \text{eval}(S_1, \overline{s}) = \overline{\circ}(\text{eval}(S_1, \overline{s_1}), \ldots, \text{eval}(S_1, \overline{s_n})), \\ \text{eval}(S_2, \overline{s}) = \overline{\circ}(\text{eval}(S_2, \overline{s_1}), \ldots, \text{eval}(S_2, \overline{s_n})), \end{cases}$$

where $=^1$ follows from Definition 3 and $\Rightarrow^4$ from Definition 10. By the inductive hypothesis,

$$\text{eval}(S_1, \overline{s_i}) \preceq \text{eval}(S_2, \overline{s_i}), \ i \in \{1, \ldots, n\},$$

and by Proposition 4(a), $\overline{\circ}$ is monotonic so that (i) holds. □

Proposition 2. (*See Section* 4.3.)

Proof. By hypothesis $C \rightsquigarrow^{S_1} C_1$ and $C \rightsquigarrow^{S_2} C_2$ so that, by Definition 11, $C_1 \in \mathcal{C}^{X_1}$ and $C_2 \in \mathcal{C}^{X_2}$ where $X_1 \subseteq X$ and $X_2 \subseteq X$ and

$$C_1 = \{c_1 \mid \exists c \in C \,.\, c \rightsquigarrow^{S_1} c_1\} \quad \text{and} \quad C_2 = \{c_2 \mid \exists c \in C \,.\, c \rightsquigarrow^{S_2} c_2\}.$$

As $S_1 \preceq S_2$, by Lemma 2,

$$\forall c_2 \in C_2 : \exists c_1 \in C_1 \quad \text{such that} \quad c_1 \preceq c_2. \tag{12}$$

By Definition 7, if $c_1 \preceq c_2$ then c_1 and c_2 are constrained on the same variable $x \in X$. Then, it follows from (12) that

$$X_2 \subseteq X_1. \tag{13}$$

Let C_{1x} and C_{2x} be the sets of constraints, in C_1 and C_2, respectively, with constrained variable $x \in X$ (note that C_{1x} and C_{2x} can be the empty set). It follows from (12) and (13) that, for each $c_2 \in C_{2x}$, there exists $c_1 \in C_{1x}$ such that $c_1 \preceq c_2$.

Suppose that c_{1x}, c_{2x}, c'_{1x}, and c'_{2x} are the constraints for $x \in X$ in the stores S_1, S_2, S'_1, and S'_2, respectively. By hypothesis $S_1 \preceq S_2$ so that, by Definition 9, $c_{1x} \preceq c_{2x}$. Since $S_1 \cup C_1 \mapsto S'_1$ and $S_2 \cup C_2 \mapsto S'_2$, by Definition 12,

$$c'_{1x} = \cap_L(C_{1x} \cup \{c_{1x}\}),$$
$$c'_{2x} = \cap_L(C_{2x} \cup \{c_{2x}\}).$$

As consequence of Definition 8 and contractance property of $\cap_L$,

$$c'_{1x} \preceq c'_{2x}, \quad \text{for each} \quad x \in X.$$

Therefore, by Definition 9, $S'_1 \preceq S'_2$. □

Lemma 3. *Let $C \in \mathcal{C}^X$ and $S, R \in \mathcal{SS}^X$. If R is a solution for $C \cup S$, then $R \preceq S$.*

Proof. By Definition 13, R is consistent, and

$$C \cup S \rightsquigarrow^R C_R \text{ and } R \cup C_R \mapsto R. \tag{14}$$

By Definition 11, if $C \cup S \rightsquigarrow^R C_R$, then C_R is equivalent to $C_R = C_1 \cup C_2$ where

$$C \rightsquigarrow^R C_1 \text{ and } S \rightsquigarrow^R C_2,$$

and also

$$C_2 = \{c' \mid \exists c \in S \,.\, c \rightsquigarrow^R c'\}.$$

Observe that if $c = x \sqsubseteq \langle \overline{s}, t\rangle$ is a simple constraint and $x \in V_L$ then, by Definition 7, $\langle \overline{s}, t\rangle \in R^s_L$ and, by Definition 5, $\overline{s} \in \overline{L^s}$ and $t \in L^s$. Therefore, by

Definition 10

$$\text{eval}(R, c) = x \sqsubseteq \langle \text{eval}(R, \overline{s}), \text{eval}(R, t) \rangle = x \sqsubseteq \langle \overline{s}, t \rangle \ = \ c.$$

By Definition 11, this means that $c \leadsto^R c$ if c is simple. Since S contains only simple constraints, then by Definition 11, $C_2 = S$.

Moreover, from (14), $R \cup C_R \mapsto R$ and, as shown previously, $C_R = C_1 \cup C_2 = C_1 \cup S$. Thus $R \cup C_1 \cup S \mapsto R$ and, by Proposition 5, $R \preceq S$. □

Theorem 1 (Correctness). (*See Section* 5.1.)

Proof. Let S_0 be the initial value of S and $C = C \cup S_0$. Suppose that there are k iterations of the *repeat* loop and that, for each i where $1 \leq i \leq k$, S_i is the value of the constraint store S at Step (5), after completing ith iterations of the *repeat* loop.

Suppose first that a solution R (for $C \cup S_0$) exists. Then, by Definition 13, R is consistent, $R \in \mathcal{SS}^X$ and

$$C \leadsto^R C_R \text{ and } R \cup C_R \mapsto R. \tag{15}$$

Note that, initially $S_0 = S \in \mathcal{SS}^X$. Then, by Lemma 3,

$$R \preceq S_0. \tag{16}$$

We show by induction on i that, after $i \geq 0$ iterations of the *repeat* loop,

$$R \preceq S_i. \tag{17}$$

The base case when $i = 0$ is given by (16). For the inductive step, suppose that there are at least $i > 0$ iterations of the repeat loop and that, after $i - 1$ steps, we have $R \preceq S_{i-1}$. Then, in the ith iteration

$$C \leadsto^{S_{i-1}} C' \qquad \text{by Step (2);} \tag{18}$$

$$S_{i-1} \cup C' \mapsto S_i, \qquad \text{by Step (4).} \tag{19}$$

It follows from (15), (18), (19), and Proposition 2 that (17) holds.

Therefore, $R \preceq S_k$. As R is consistent, by Proposition 6, S_k is consistent. However, the procedure terminates before the $k+1$th iteration so that the test in Step (5) is true and we must have $S_k = S_{k-1}$. By (18) and (19)

$$C \leadsto^{S_{k-1}} C' \text{ and } S_{k-1} \cup C' \mapsto S_k. \tag{20}$$

Thus, by Definition 13, S_k is a solution for C (i.e., $C \cup S_0$). Moreover, if R is another solution for $C \cup S_0$, then as $R \preceq S_k$ and as R was any solution for $C \cup S_0$, $S_k = \text{mgs}(C \cup S_0)$.

Suppose next that there is no solution for $C \cup S_0$. Then $S_{k-1} \neq S_k$ or else, by (20), S_k would be a solution. Thus, in this case, as the procedure terminates before the $k+1$th iteration so that the test in Step (5) is true, we must have S_k is inconsistent. □

Proposition 7. *$precision_L$ is strict monotonic. That is,*

$$precision_L(c) \prec precision_L(c') \text{ if } c \prec c'.$$

PROOF. Suppose that $c_1 = x \sqsubseteq \langle \overline{a_b}, c_d \rangle$ and $c_2 = x \sqsubseteq \langle \overline{a'_{b'}}, c'_{d'} \rangle$ where $c_1 \prec c_2$. Then, we have to prove that

$$precision_L(c_1) \prec precision_L(c_2),$$

which, by Definition 14, is equivalent to showing

$$(\hat{a} \diamond_L c, b \diamond_B d) \prec (\hat{a'} \diamond_L c', b' \diamond_B d'). \tag{21}$$

By hypothesis, $c_1 \prec c_2$, so that by Definition 7

$$\langle \overline{a_b}, c_d \rangle \prec \langle \overline{a'_{b'}}, c'_{d'} \rangle$$

and, by the product of lattices (i.e., direct product), either

$$\overline{a_b} \prec \overline{a'_{b'}} \quad \text{and} \quad c_d \preceq c'_{d'} \text{ or} \tag{22}$$

$$\overline{a_b} \preceq \overline{a'_{b'}} \quad \text{and} \quad c_d \prec c'_{d'}. \tag{23}$$

If (22) holds, then, by the product of lattices (i.e., lexicographic product),

$$(\hat{a} \prec \hat{a'} \text{ or } \hat{a} = \hat{a'} \text{ and } b \prec b') \quad \text{and} \quad (c \prec c' \text{ or } c = c' \text{ and } d \preceq d'). \tag{24}$$

Similarly, if (23) holds, then,

$$(\hat{a} \prec \hat{a'} \text{ or } \hat{a} = \hat{a'} \text{ and } b \preceq b') \quad \text{and} \quad (c \prec c' \text{ or } c = c' \text{ and } d \prec d'). \tag{25}$$

Therefore, $\hat{a} \prec \hat{a'}$ and $c \preceq c'$, $\hat{a} = \hat{a'}$ and $c \prec c'$, or $\hat{a} = \hat{a'}$ and $c = c'$. However, we have

$$\left\{ \begin{array}{l} \hat{a} \prec \hat{a'} \text{ and } c \preceq c' \\ \hat{a} = \hat{a'} \text{ and } c \prec c' \end{array} \right\} \Rightarrow^1 \hat{a} \diamond_L c \prec \hat{a'} \diamond_L c' \Rightarrow^2 \text{ (21) holds,}$$

$$\hat{a} = \hat{a'} \text{ and } c = c' \left\{ \begin{array}{l} \Rightarrow^3 b \prec b' \text{ and } d \preceq d' \\ \Rightarrow^4 b \preceq b' \text{ and } d \prec d' \end{array} \right\} \Rightarrow^{1,5}$$

$$\hat{a} \diamond_L c = \hat{a'} \diamond_L c' \text{ and } b \diamond_B d \prec b' \diamond_B d' \Rightarrow^2 \text{ (21) holds,}$$

where $\Rightarrow^1$ follows from strict monotonicity of $\diamond_L$ in Definition 14, $\Rightarrow^2$ from the product of lattices (i.e., lexicographic product), $\Rightarrow^3$ from (24), $\Rightarrow^4$ from (25), and $\Rightarrow^5$ from strict monotonicity of $\diamond_B$ in Definition 14. □

THEOREM 2 (TERMINATION). (*See Section* 5.2.)

PROOF. Let S_0 be the initial value of S. Suppose there are at least $i \geq 0$ iterations of the repeat loop. If $i \geq 1$, let $S_i \in \mathcal{SS}^X$ be the constraint store at the end of the ith iteration. Suppose first that S_j is inconsistent for some j, $0 \leq j \leq i$. Then either the test in Step (0) (if $j = 0$) or the test in Step (5) (if $j > 0$) fails and the procedure terminates after $i = j$ iterations. We now assume that S_i is consistent for all $i \geq 0$ such that S_i is defined.

For each $i \geq 0$ such that S_i is defined and each $x \in X \cap V_L$ and $L \in \mathcal{L}$, let c_x^i denote the simple (consistent) constraint in S_i. Also, for each $x \in X$ and some $L \in \mathcal{L}$, let $precision_L(c_x^i) = (\phi_x^i, \psi_x^i)$. For each $i \geq 0$ such that S_i is defined, let

$$X_i = \left\{ x \in X \;\middle|\; \phi_x^i = \top_{\Re^+} \right\},$$
$$Y_i = \left\{ y \in X \;\middle|\; \phi_y^i \prec \top_{\Re^+} \right\}.$$

In the ith iteration we have, by Step (4), $S_{i-1} \cup C' \mapsto S_i$ so that, by Proposition 5, $S_i \preceq S_{i-1}$. Thus, by Definition 9, for each $x \in X$, $c^i_x \preceq c^{i-1}_x$, and, by Proposition 7, $(\phi^i_x, \psi^i_x) \preceq (\phi^{i-1}_x, \psi^{i-1}_x)$. As the order is lexicographic, we have for all $x \in X$, $\phi^i_x \leq \psi^{i-1}_x$. Thus $X_i \subseteq X_{i-1}$ and, if $X_i = X_{i-1}$, then $Y_i = Y_{i-1}$. If *no_difference*$_\delta(S_{i-1}, S_i)$ holds, then the test in Step (5*) is true and the procedure terminates. We now assume that the termination condition *no_difference*$_\delta(S_{i-1}, S_i)$ does not hold. There are two cases:

(1) $X_i \subset X_{i-1}$;
(2) $X_i = X_{i-1}$, $Y_i = Y_{i-1}$ and $\exists y \in Y_i \,.\, (\phi^{i-1}_y, \psi^{i-1}_y) - (\phi^i_y, \psi^i_y) > (\varepsilon, 0)$.

As $X_i \subset X_{i-1}$ can occur at most $\#X_0$ different values for i, we can assume that, for some iteration $j \geq 0$, $X_i = X_j$ for all $i \geq j$ and, for all $i > j$, Case (2) applies. Thus, for all $i > j$, $Y_i = Y_{i-1}$, so that, as $\phi^i_x \leq \psi^{i-1}_x$ for all $x \in Y_i$,

$$\sum_{x \in Y_i} \phi^i_x \leq \sum_{x \in Y_{i-1}} \phi^{i-1}_x.$$

At Step (5*) of the ith iteration where $i > j$, either the repeat loop terminates or *no_difference*(S_{i-1}, S_i) does not hold and thus, by Case (2),

$$\sum_{x \in Y_{i-1}} \phi^{i-1}_x - \sum_{x \in Y_i} \phi^i_x > \varepsilon.$$

Let

$$k = \left\| \sum_{x \in Y_j} \phi^j_x / \varepsilon \right\| + j,$$

where $\|r\|$ denotes the integer part of $r \in \Re^+$. It follows that, if there is a kth iterations of the repeat loop, then $\sum_{x \in Y_k} \phi^k_x < \varepsilon$ and hence *no_difference*(S_{k-1}, S_k) holds. Thus the procedure has at most k iterations. □

THEOREM 3 (CORRECTNESS IN THE EXTENDED SCHEMA). (*See Section* 5.2.)

PROOF. Suppose the procedure terminates after k iterations with S_ε the final value of the constraint store S. It has already been shown in (17) of the proof of Theorem 1 that

$$R \preceq S_\varepsilon.$$

Thus, for all $c_x \in R$ and $c'_x \in S_\varepsilon$, *precision*$(c_x) \preceq$ *precision*(c'_x). Let $\delta \in \Re\mathcal{I}$ be the maximum of *precision*$(c'_x) -$ *precision*(c_x) for all $x \in X$. Then *no_difference*$_\delta(S_\varepsilon, R)$ holds. Thus, by Definition 16, S_ε is an approximate solution for $C \cup S$. □

THEOREM 4. (*See Section* 5.2.)

PROOF. In the previous proof we have $R \preceq S_\varepsilon$ for any $\varepsilon \geq 0.0$. Therefore $R \preceq S_{\varepsilon_1}$ and $R \preceq S_{\varepsilon_2}$ Thus, we just have to show that $S_{\varepsilon_1} \preceq S_{\varepsilon_2}$.

Suppose the procedures solve$_{\varepsilon_2}(C, S)$ and solve$_{\varepsilon_1}(C, S)$ terminate after k_2 and k_1 iterations, respectively. Therefore, the check, in Line (5*), for the repeat loop for steps from 1 to k_2 must also succeed for ε_1. Thus $k_1 \geq k_2$.

Suppose that S_i is the value of S at the end of the ith iteration of the repeat loop ($1 \le i \le k_1$) (so that $S_{k_1} = S_{\varepsilon_1}$ and $S_{k_2} = S_{\varepsilon_2}$). We show, by induction on i, where $k_2 \le i \le k_1$, that

$$S_i \preceq S_{\varepsilon_2}. \tag{26}$$

The base case when $i = k_2$ is obvious. For the inductive step, suppose that $i > k_2$ and assume that $S_{i-1} \preceq S_{\varepsilon_2}$. By Step (4) of the extended operational schema, we have

$$S_{i-1} \cup C' \mapsto S_i \tag{27}$$

at the end of the ith iteration of the repeat loop. Thus, by Proposition 5, $S_i \preceq S_{i-1}$ so that (26) holds. Therefore, letting $i = k_1$ we obtain $S_{\varepsilon_1} \preceq S_{\varepsilon_2}$. □

ACKNOWLEDGMENTS

We would like to thank the referees of earlier versions of this paper. Their comments have helped us improve many aspects of this work.

REFERENCES

AÏT-KACI, H. 1999. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, MA.

ALLEN, J. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11, 832–843.

APT, K. 1999. The rough guide to constraint propagation. In *5th International Conference on Principles and Practice of Constraint Programming (CP'99, Alexandria, Virginia)*, J. Jaffar, Ed. Lecture Notes in Computer Science, vol. 1713. Springer-Verlag, Berlin, Germany, 1–23.

APT, K. 2000. The role of commutativity in constraint propagation algorithms. *ACM Trans. Programm. Lang. Syst. 22*, 6, 1002–1036.

BAADER, F. AND SCHULZ, K. 1995. On the combination of symbolic constraints, solution domains and constraints solvers. In *1st International Conference on Principles and Practice of Constraint Programming (CP'95, Cassis, France)*, U. Montanari and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 976. Springer-Verlag, Berlin, Germany, 380–397.

BARTH, P. AND BOCKMAYR, A. 1996. Modelling 0-1 problems in CLP($\mathcal{PB}$). In *2nd International Conference on the Practical Application of Constraint Technology (PACT'96)*, M. Wallace, Ed. Prolog Management Group, London, U.K., 1–9.

BENHAMOU, F. 1995. Interval constraint logic programming. In *Constraint Programming: Basics and Trends*, A. Podelski, Ed. Lecture Notes in Computer Science, vol. 910. Springer-Verlag, Berlin, Germany, 1–21.

BENHAMOU, F., GOUALARD, F., GRANVILLIERS, L., AND PUGET, J.-F. 1999. Revising hull and box consistency. In *16th International Conference on Logic Programming (ICLP'99, Las Cruces, New Mexico,)* D. De Schreye, Ed. The MIT Press, Cambridge, MA, 230–244.

BENHAMOU, F. AND OLDER, W. 1997. Applying interval arithmetic to real, integer and Boolean constraints. *J. Logic Programm. 32*, 1 (July), 1–24.

BIRKHOFF, G. 1967. *Lattice Theory*, 3rd ed. Coloquium Publications, vol. XXV. American Mathematical Society, Providence RI.

BISTARELLI, S., MONTANARI, U., AND ROSSI, F. 1995. Constraint solving over semirings. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95, Québec, Canada)*. Morgan Kaufman, San Francisco, CA, 624–630.

CARLSSON, M., OTTOSSON, G., AND CARLSON, B. 1997. An open-ended finite domain constraint solver. In *9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97, Southampton, U.K.)*, U. Montanari and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 1292. Springer-Verlag, Berlin, Germany, 191–206.

CLEARY, J. 1987. Logical arithmetic. *Fut. Comput. Syst. 2*, 2, 125–149.

CODOGNET, P. AND DIAZ, D. 1993. Boolean constraint solving using clp(FD). In *1993 International Symposium on Logic Programming (ILPS'93 Vancouver, British Columbia, Canada)*, D. Miller, Ed. The MIT Press, Cambridge, MA, 525–539.

CODOGNET, P. AND DIAZ, D. 1994. clp(B): combining simplicity and efficiency in Boolean constraint solving. In *6th International Symposium on Programming Languages Implementation and Logic Programming (PLILP'94, Madrid, Spain)*. Lecture Notes in Computer Science, vol. 844. Springer-Verlag, Berlin, Germany, 244–260.

CODOGNET, P. AND DIAZ, D. 1996a. Compiling constraints in clp(FD). *J. Logic Programm. 27*, 3, 185–226.

CODOGNET, P. AND DIAZ, D. 1996b. A simple and efficient Boolean solver for constraint logic programming. *J. Automat. Reason. 17*, 1, 97–129.

DAVEY, B. AND PRIESTLEY, H. 1990. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, England.

DIAZ, D. AND CODOGNET, P. 1993. A minimal extension of the WAM for clp(FD). In *10th International Conference on Logic Programming (ICLP'93, Budapest, Hungary)*, D. Warren, Ed. The MIT Press, Cambridge, MA, 774–790.

DIAZ, D. AND CODOGNET, P. 2001. Design and implementation of the GNU prolog system. *J. Funct. Logic Programm. 2001*, 6 (Oct.).

FERNÁNDEZ, A. 2000. clp(L) version 0.21, user manual. Available online at http://www.lcc.uma.es/∼afdez/generic.

FERNÁNDEZ, A. 2002. A generic, collaborative framework for interval constraint solving. Ph.D. dissertation, E.T.S.I.I., Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain. English version available online at http://www.lcc.uma.es/∼afdez/papers.html.

FERNÁNDEZ, A. AND HILL, P. 1999a. Interval constraint solving over lattices using chaotic iterations. In *ERCIM/COMPULOG Workshop on Constraints*, K.Apt, A. Kakas, E. Monfroy, and F. Rossi, Eds. Dept., of Computer Science, University of Cyprus, Paphos, Cyprus. Available online at http://www.cwi.nl/ERCIM/WG/Constraints/Workshops/Workshop4/Program.

FERNÁNDEZ, A. AND HILL, P. 1999b. An interval lattice-based constraint solving framework for lattices. In *4th International Symposium on Functional and Logic Programming (FLOPS'99, Tsukuba, Japan)*, A. Middeldorp and T. Sato, Eds. Lecture Notes in Computer Science, vol. 1722. Springer-Verlag, Berlin, Germany, 194–208.

FERNÁNDEZ, A. AND HILL, P. 2000. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Constraints 5*, 3, 275–301.

FRÜHWIRTH, T. 1998. Theory and practice of constraint handling rules. *J. Logic Programm. 37*, 95–138.

GEORGET, Y. AND CODOGNET, P. 1998. Compiling semiring-based constraints with clp(FD,S). In *4th International Conference on Principles and Practice of Constraint Programming (CP'98, Pisa, Italy)*, M. Maher and J.-F. Puget, Eds. Lecture Notes in Computer Science, vol. 1520. Springer-Verlag, Berlin, Germany, 205–219.

GERVET, C. 1997. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints 1*, 3, 191–244.

GOUALARD, F., BENHAMOU, F., AND GRANVILLIERS, L. 1999. An extension of the WAM for hybrid interval solvers. *J. Funct. Logic Programm. 1999*, 1 (April), 1–36. Special issue of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages.

HICKEY, T. 2000. CLIP: a CLP(Intervals) dialect for metalevel constraint solving. In *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000, Boston, Massachusetts)*, E. Pontelli and V. Costa, Eds. Lecture Notes in Computer Science, vol. 1753. Springer-Verlag, Berlin, Germany, 200–214.

HOFSTEDT, P. 2000. Better communication for tighter cooperation. In *1st International Conference on Computational Logic (CL'2000, London, U.K.)*, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer-Verlag, Berlin, Germany, 342–357.

IF/PROLOG. 1994. *IF/Prolog V5.0A, Constraints Package*. Siemens Nixdorf Informationssysteme AG, Munich, Germany.

ISO/IEC. 1995. *ISO/IEC 13211-1: 1995 Information Technology—Programming Languages—Prolog—Part 1: General Core*. International Standard Organization, Geneva, Switzerland.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. The CLP($\Re$) language and system. *ACM Trans. Programm. Lang. Syst. 14*, 3, 339–395.

Le Provost, T. and Wallace, M. 1993. Generalized constraint propagation over the CLP scheme. *J. Logic Programm. 16*, 3 (Aug.), 319–359.

Lee, J. and van Emden, M. 1993. Interval computation as deduction in CHIP. *Journal Logic Programm.* (Special Issue on Constraint Logic Programming) *16*, 3-4, 255–276.

Moore, R. 1966. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ.

N'Dong, S. 1997. Prolog IV ou la programmation par contraintes selon PrologIA. In *Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JF-PLC'97)*. Edition HERMES, Orléans, France, 235–238.

Older, W. 1989. Interval arithmetic specification. Tech. Rep. Bell-Northern, Research Computing Research Laboratory, Ottawa, Ontario, Canada.

Older, W. and Vellino, A. 1993. Constraint arithmetic on real intervals. In *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmesauer, Eds. The MIT Press, Cambridge, MA, 175–195.

Refalo, P. and Van Hentenryck, P. 1996. CLP($\Re_{lin}$) revised. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96, Bonn, Germany)*, M. Maher, Ed. The MIT Press, Cambridge, MA, 22–36.

Schiex, T., Fargier, H., and Verfaillie, G. 1995. Valued constraint satisfaction problems: Hard and easy problems. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95, Québec, Canada)*. Morgan Kaufman, San Francisco, CA, 631–637.

Sicstus Manual. 1994. *SICStus Prolog User's Manual, Release* 3#5. The Intelligent Systems Laboratory, Swedish Institute of Computer Science. Web site: http://www.sics.se.

Sidebottom, G. and Havens, W. 1992. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computat. Intell. 8*, 4, 601–623.

Slavík, V. 1986. A note on the lattice of intervals of a lattice. *Algebra Universalis 23*, 1, 22–23.

Van Hentenryck, P., Saraswat, V., and Deville, Y. 1998. Design, implementation and evaluation of the constraint language cc(FD). *J. Logic Programm. 37*, 1–3 (Oct.), 139–164.

Walinsky, C. 1989. CLP(Σ^*): Constraint logic programming with regular sets. In *6th International Conference on Logic Programming (ICLP'89, Lisbon, Portugal)*, G. Levi and M. Martelli, Eds. The MIT Press, Cambridge, MA, 181–196.

Received December 2000; revised March 2002; accepted March 2003