Constraint Propagation on Multiple Domains

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

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

Abstract In previous work we have presented a simple generic framework to solve constraints on any domain (finite or infinite) which has a lattice structure. The approach is based on the use of a single constraint similar to the indexicals used by constraint logic programming (CLP) over finite domains and on a particular definition of an interval lattice built from any computation domain.
In this paper we generalise this work by providing a constraint propagation mechanism that allows the flow of information between different computation domains. This generalisation is done by letting the constraint operators be defined over more than one domain and describing new generic constraint operators that are defined for any user or system domain.
We show, by means of examples, how this extension improves the expressiveness of the system and addresses novel proposals for constraint propagation.
Keywords: Lattices, constraint solving, indexicals, solvers.

1 Introduction

Most of the existing CLP languages only provide support for solving constraints on specified domains. This implies that the constraints are usually restricted to just values in the given computation domain. For example an arithmetic constraint such as $x + y = z$ must be defined so that all the elements x, y and z belong to the same domain (i.e. integer or real). However, in practice, constraints are often not specific to any given domain. Therefore the formulation of real problems has to be artificially adapted to a domain that is supported by the system. Many problems are most naturally expressed using heterogeneous constraints, involving more than one domain. For example, consider a community of people. To write a constraint that determines the set of people that are taller than two meters cannot be directly coded in most existing CLP languages. The main exception to this is the CHR language [10] that allows for user-defined domains.

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

In [7], we have proposed an alternative to the CHR approach that is based on the indexical approach of clp(FD) [4] which is known to be highly efficient [8]. In this proposal, we defined a single framework for solving constraints on any domain with a lattice structure. Thus, this framework allows constraint solving on all the classical domains such as Booleans, reals, finite ranges of integers and sets as well as new specialised domains designed for specific applications. In this paper we generalise this work by allowing the cooperation of solvers defined on different domains. Constraints can be propagated from one computation domain to another thereby allowing information to flow between the different domains. Moreover, the basic framework allows new domains to be constructed from existing domains by means of lattice combinators. Thus the generalisation described here allows constraint propagation to go from the basic domains to the combined domains and vice-versa. For example, consider the direct product domain $community = \Re \times string$ (where $string$ is a set of names) to define a community of people ordered lexicographically by their height (first argument) and name (second argument). Assume that a constraint $>'/2$ on the domain $community \times \Re$ is also defined as $(name, high) >' limit \Leftrightarrow high >_{\Re} limit$. Then, the constraint $person >' 2.00$ where $person \in community$ determines the set of people higher than two meters.

The paper is structured as follows. In Section 2, some considerations about related work are shown. Section 3 recalls some algebraic concepts and Section 4 generalises our previous work to allow for multiple domains. In Section 5, we show how to define generic constraints. Sections 6, 7 and 8 develop non-trivial examples to show the expressivity of our generalised framework. The paper ends with the conclusions and some considerations about further work.

2 Related Proposals

Interval arithmetic [3] is a useful and versatile technique to solve constraint satisfaction problems. Computation on intervals is widely used for the solving of CLP programs [1, 2, 11]. Traditionally, it has been applied to numeric domains (i.e. floating point numbers) and the techniques approximate a computed real number with the closest floating point number smaller (resp. higher) than the computed number for the lower (resp. upper) bound of an interval. The concept of approximation to a computed value is intended for numeric domains and cannot be easily generalised to non-numeric domains supported by our framework.

In the clp(FD) model proposed in [4] for the finite domain (FD), a range is defined as a subset of $\{0, 1, ..., infinity\}$ where $infinity$ is a particular integer denoting the greatest value that a variable can take. The notation a..b is used as a shorthand for the set $\{x \mid a \leq x \leq b\}$ (that is, [a,b] in interval notation). The clp(FD) model is based on a unique constraint $x\ in\ a..b$ that forces a FD variable x to have values in the range $a..b$. Narrowing is executed by the intersection of ranges associated to the same variable. Constraint propagation is executed via the functions $max/1$ and $min/1$ (also $dom/1$ but we do not treat it here) that receive a FD variable, i.e. x, as argument and return, respectively, the maximum and minimum value associated to the variable by a constraint $x\ in\ r$, that is, the maximum and minimum in r. For example, the following constraints $x\ in\ 4..10$, $x \neq 10$, $y\ in\ 0..max(x)$ leads to the constraint $y\ in\ 0..9$ since $x\ in\ 4..10$, $x \neq 10$ results in $x\ in\ 4..9$ and $max(x) = 9$. We note that the clp(FD) model is not valid for continuous domains. For example, consider the real domain, the constraints $x\ in\ 4.0..10.0$, $x \neq 10.0$, $y\ in\ 0.0..max(x)$ leads to the wrong constraint $y\ in\ 0.0..10.0$ since the constraints $x\ in\ 4.0..10.0$, $x \neq 10.0$ cannot be expressed in the clp(FD) model and thus $max(x) = 10.0$.

In [7], we have generalised both the interval reasoning and the clp(FD) approach to allow for any domain (even continuous) with lattice structure. Given an underlying (original) computation domain L, we consider, for constraint propagation, two computation domains constructed from L: a left-side computation domain and a right-side computation domain (in an interval). These domains are constructed by the combination of the elements in L with the usual open and closed brackets used in interval notation. A key feature and novelty of our approach is that the bracket is part of the value to be propagated. We do this by combining each element of the domain L with one of two brackets '**)**' or ']' to create a new domain L_r (called the right-side computation domain). The domain L_r is used for constraint propagation in the right bound of an interval. Similarly, for the left bound of an interval, we define the domain L_l (called the left-side computation domain) resulting of combining each element of L with one of two left-side bracket,'**(**' or '['. Now each element in L is represented, at least, by one element in L_r or L_l. For example, consider the real domain (that is $L = \Re$), the real value 1.0 is represented by the value (1.0,']') in $\Re_r$. For clarity, this value is denoted as **1.0**]. Also , the greatest value smaller than 1.0 is represented by the value (1.0,')') in $\Re_r$ and denoted as **1.0)**, whereas the lowest value higher than 1.0 is represented by the value ('(',1.0) in $\Re_l$ and denoted as **(1.0**. The greatest (resp., least) possible value for a variable is denoted by $max(x)$ (resp., $min(x)$). Both propagation of constraints and constraint narrowing are executed on these domains. For instance, the constraints

$$x\ in\ [\mathbf{4.0..10.0}], \quad x \neq \mathbf{10.0}], \quad y\ in\ [\mathbf{0.0}..max(x)$$

lead to the constraint $y\ in\ [\mathbf{0.0..10.0})$ since the first two of the constraints are equivalent to the constraint [1] $x\ in\ [\mathbf{4.0..10.0})$ so that $max(x) = \mathbf{10.0})$. Note that $max(x) \in \Re_r$.

Domains L_r and L_l are also very important to determine the monotonicity of constraints (see Section 4.4).

3 Preliminaries

The notation $card(S)$ denotes the cardinality of a set S. Let C be an ordered set. Then $\preceq_C$ denotes the ordering relation on C. We write $c \sim c'$ if either $c \preceq_C c'$ or $c' \preceq_C c$ and $c \not\sim c'$ otherwise. An element s in C is a *lower (upper)*

[1] Note that the value **10**], that belongs to $\Re_r$, has been removed

bound of a subset $E \subseteq C$ if and only if $\forall x \in E$: $s \preceq x$ ($x \preceq s$). If the set of lower (upper) bounds of E has a greatest (least) element, then that element is called the *greatest lower bound* (*least upper bound*) of E and denoted by $glb_C(E)$ ($lub_C(E)$). For simplicity, we write $glb_C(x, y)$ and $lub_C(x, y)$ when E contains only two elements x and y. L is a *lattice* if it is an ordered set and, for any two elements $x, y \in L$, $lub_L(x, y)$ and $glb_L(x, y)$ exist. $glb_L(L)$, if it exists, is called *the bottom element* of L and written $\bot_L$. Similarly, $lub_L(L)$, if it exists, is called *the top element* of L and written $\top_L$. The lack of a bottom or top element can be remedied by adding a fictitious one. Thus, the *lifted lattice* of L is $L \cup \{\bot_L, \top_L\}$ where, if $glb_L(L)$ (resp. $lub_L(L)$) does not exist, $\bot_L$ (resp. $\top_L$) is a new element not in L such that $\forall a \in L$, $\bot_L \prec a$ (resp. $a \prec \top_L$). The *dual* $\overline{L}$ *of* a lattice L, is the lattice that contains exactly the same elements as L and obtained by interchanging $glb_L(a, b)$ and $lub_L(a, b)$ for any $a, b \in L$.

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

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

lub is defined dually to glb[2]*.*

4 Theoretical work

In [7] the generic framework was defined over any single domain with lattice structure. Here, we present a generalised form of this construction that allows for multiple domains.

4.1 The computation domain

We assume that there is a set $\mathcal{L}$ of n (lifted) lattice-structure computation domains with n (possibly infinite) sets of variables V_L associated with each of the domains L in $\mathcal{L}$. We define the domain of brackets as $B = \{$ ')', ']' $\}$ where B is a lattice with the ordering ')' $\prec_B$ ']'. We let $\}$ denote any element of B. In next sections we use the functions $min_B/2$ and $min'_B/2$ on B. $min_B(\}_1, \}_2)$ returns the smallest value between brackets $\}_1$ and $\}_2$ in B and $min'_B(\}_1, \}_2)$ returns ')' if $\}_1 \neq \}_2$ and ']' otherwise. For each $L \in \mathcal{L}$, we also define

$$\begin{aligned}
L_l &= \{\ (l, b) \mid l \in \overline{L} \wedge b \in B\} &\quad& \text{(left-side computation domain for } L) \\
L_r &= \{\ (l, b) \mid l \in L \wedge b \in B\} && \text{(right-side computation domain for } L)
\end{aligned}$$

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

For simplicity, any element $(e, \})\in L_l$ (resp. $(e, \})\in L_r$) is written as $\{\mathbf{e}$ (resp. $\mathbf{e}\}$). For example, if $a \in \overline{L}$ then $(a,`)')$ belongs to $L_l$ and is denoted as $(\mathbf{a}$. Also, if $a \in L$ then $(a,`)')$ belongs to L_r and is denoted as $\mathbf{a})$. For example, $(\mathbf{3} \in Integer_l$, $\mathbf{3}) \in Integer_r$, $[\mathbf{4.67} \in \Re_l$ and $\mathbf{2.33}) \in \Re_r$.

L_l and L_r are lattices under the lexicographic ordering $(\overline{L}, B)$ and (L, B) respectively. For example, by simplicity consider L to be the (lifted) domain of naturals. Then

$$
\begin{array}{ll}
0) \ < 0] \ < 1) \ < 1] < \ldots < \top_L) \ < \top_L] & \text{(On } L_r) \\
(\top_L \ < [\top_L \ < \ldots < (1 \ < [1 < (0 \ < [0 & \text{(On } L_l)
\end{array}
$$

where $\top_L$ denotes the ficticial top element for the natural domain. We also let $\mathcal{L}_l$ denote the set $\{L_l \mid L \in \mathcal{L}\}$, $\mathcal{L}_r$ the set $\{L_r \mid L \in \mathcal{L}\}$ and $\mathcal{L}^* = \mathcal{L}_l \cup \mathcal{L}_r$.

4.2 Operators

For all $L \in \mathcal{L}$, we allow the definition of operators on the domains L_l and L_r. A n-ary operator $\circ_l$ (for L_l) is an operator defined as $\circ_l :: L_1 \times \ldots \times L_n \to L_l$ where $L_i \in \mathcal{L}^*$ $(1 \leq i \leq n)$. Analogously a n-ary operator $\circ_r$ (for L_r) is an operator defined as $\circ_r :: L_1 \times \ldots \times L_n \to L_r$ where $L_i \in \mathcal{L}^*$ $(1 \leq i \leq n)$.

For example, Table 1 (see page 15) shows the definitions of the operator $trunc_l$ (for $Integer_l$) and $trunc_r$ (for $Integer_r$) where $\{\mathbf{a} \in \Re_l$, $\mathbf{a}\} \in \Re_r$ and $trunc(\mathbf{a})$ is defined to return the integer part of the real value $\mathbf{a}$. For example, $trunc_l([\mathbf{3.2}) = [\mathbf{3}$ and $trunc_r(\mathbf{4.1})) = \mathbf{4})$.

Definition 1. *(Monotonicity of operators) Let $\circ' :: L_1 \times \ldots \times L_n \to L'$ be a n-ary constraint operator for L' where $L' \in \{L_l, L_r\}$ and $L \in \mathcal{L}$. Then $\circ'$ is* monotonic *if, for all $t_i, t'_i \in L_i$ and $i \in \{1, \ldots, n\}$ such that $t_i \preceq_{L_i} t'_i$ we have*

$$\circ'(t_1, \ldots, t_i, \ldots, t_n) \ \preceq_{L'} \circ'(t'_1, \ldots, t'_i, \ldots, t'_n)$$

We impose the condition, called the *law of monotonicity for operators*, that all the operators must be monotonic. For example, Table 1 shows, for some $L, L' \in \mathcal{L}$ (and probably $L = L'$), the definition of the operator $-_r$ (for L_r). For simplicity, suppose that $L' = L$ (in next sections we show the motivation to define the first argument on the domain L'_r). Note that the second argument of this operator is declared on L_l. For example, consider $L = Integer$ and the expression $20] -_r x$. Then x cannot belong to L_r since as x increases on L_r the expression $20] -_r x$ decreases. Note, for instance, that $3] < 4] < 5]$ on $Integer_r$ but $20] -_r 3] > 20] -_r 4] > 20] -_r 5]$. The expression $20] -_r x$ is monotonic for all $x \in Integer_l$ since as x increases (resp.decreases) on L_l the expression $20] -_r x$ also increases (resp.decreases). Therefore, to satisfy the monotonicity condition, in Table 1, this operator has been declared as $-_r :: L'_r \times L_l \to L_r$.

We provide more justification for imposing the monotonicity law at the end of this section. Note, however, that all the operators used in this paper are monotonic.

Observe that, in Table 1, $+_l$, $+_r$, $-_l$ and $-_r$ depend on the definition of the operators $+$ and $-$ in the original computation domain. For example, let $L \in \{Integer, \Re\}$ and $L = L'$, and consider the usual definition of $+$ and $-$ for reals and integers. Then $\mathbf{(3}+_l\mathbf{[4} = \mathbf{(7}$ and $\mathbf{50]}+_r\mathbf{11]} = \mathbf{61]}$; also $\mathbf{(2.50}+_l\mathbf{(1.00} = \mathbf{(3.50}$ and $\mathbf{2.00]} +_r \mathbf{4.11)} = \mathbf{6.11)}$. Also, $\mathbf{(7} -_l \mathbf{4]} = \mathbf{(3}$ and $\mathbf{6.11)} -_r \mathbf{(4.11} = \mathbf{2.00]}$.

4.3 Constraints

For all $L \in \mathcal{L}$ we define the *interval domain* to be the direct product domain $R_L = \langle L_l, L_r \rangle$. Note R_L is a lattice. For simplicity, any element $\langle \mathbf{l,r} \rangle \in R_L$ is denoted as $\mathbf{l,r}$ where $\mathbf{l} \in L_l$ and $\mathbf{r} \in L_r$. For example, the element $\langle \mathbf{(1,`]')}, \mathbf{(4,`]')} \rangle$ belonging to $R_{Integer}$ is just denoted as $\mathbf{[1,4]}$ since $\mathbf{(1,`]')}$ is represented as $\mathbf{[1}$ in $Integer_l$ and $\mathbf{(4,`]`)}$ is represented as $\mathbf{4]}$ in $Integer_r$.

The ordering on R_L is $\langle l_1, r_1 \rangle \ \leq_{R_L} \ \langle l_2, r_2 \rangle \Leftrightarrow (l_1 \ \leq_{L_l} \ l_2) \wedge (r_1 \ \leq_{L_r} \ r_2)$. On interval notation, this ordering means the interval inclusion. For example, if $L_1 = integer$ then $\mathbf{[2,6)} <_{R_{L_1}} \mathbf{[1,7]}$ since $\mathbf{[2} <_{Integer_l} \mathbf{[1}$ and $\mathbf{6)} <_{Integer_r} \mathbf{7]}$.

Simple constraints. A simple interval constraint is an expression $x \sqsubseteq r$ that associates a variable $x \in V_L$ (called *the constrained variable*), for some $L \in \mathcal{L}$, to an element $r \in R_L$. For example, $x \sqsubseteq \mathbf{[1,4)}$ is a simple constraint on the integer domain whereas $r \sqsubseteq \mathbf{(1.22,4.56)}$ is a simple constraint on the real domain.

Constraint narrowing. Constraint narrowing is basically executed via an intersection rule for simple interval constraints defined on the same variable. Let $c_1 = x \sqsubseteq r_1$ and $c_2 = x \sqsubseteq r_2$ where $r_1 = \mathbf{\{a,b\}}$, $r_2 = \mathbf{\{c,d\}}$ and $a, b, c, d \in L$ for some $L \in \mathcal{L}$. Then its intersection is defined via rule $\cap_L$:

$$c_1 \ \cap_L \ c_2 = x \sqsubseteq glb_{R_L}(r_1, r_2) = \ x \sqsubseteq glb_{L_l}(\mathbf{\{a}, \mathbf{\{c}), glb_{L_r}(\mathbf{b\}}, \mathbf{d\}})$$

For examples,

$$x \sqsubseteq [2,30) \ \cap_L x \sqsubseteq (1,20] = x \sqsubseteq [2,20), \text{ in the integer domain and}$$
$$r \sqsubseteq (50.67,100.11] \ \cap_L r \sqsubseteq [55.56,150.98] = r \sqsubseteq [55.56,100.11], \text{ in } \Re$$

Other non-trivial examples are shown in [9].

Indexicals. An *indexical term* is an expression of the form $min(x)$ or $max(x)$ where $x \in V_L$ for some $L \in \mathcal{L}$. We define an overloaded evaluation function $eval_c/1$ that evaluates the indexical terms $min(x)$ and $max(x)$ with respect to a simple constraint c. Let $c = x \sqsubseteq t_l, t_r$ be a simple constraint defined on L, then $eval_c(min(x)) = t_l$ and $eval_c(max(x)) = t_r$. Note that, for $x \in V_L$, with $L \in \mathcal{L}$, the evaluation of the term $min(x)$ (resp. $max(x)$) returns a value in L_l (resp. L_r). For example, let $c = x \sqsubseteq (3.45, 6.78]$ be a constraint defined on the real domain, then $eval_c(min(x)) = \mathbf{(3.45}$ and $eval_c(max(x)) = \mathbf{6.78]}$.

Non-simple interval constraints. A non-simple interval constraint is an expression $x \sqsubseteq e$ that associates a variable $x \in V_L$ to an expression e that uses indexical terms and is evaluated[3] on R_L. For example, $x \sqsubseteq min(y), max(y)$ or $y \sqsubseteq [\mathbf{1}, max(z) +_r max(x)$ are non-simple constraints with $x, y, z \in V_{Integer}$.

Consistency of simple constraints. A simple constraint $x \sqsubseteq \ \{_1\mathbf{a},\mathbf{b}\}_2$ is *inconsistent* if
(1) $\mathbf{a}\}_1 \not\preceq_{L_r} \mathbf{b}\}_2$ (note that this means that it is inconsistent if $\mathbf{a}\}_1 \not\prec_{L_r} \mathbf{b}\}_2$ or (2) $a = b$ and $\}_1 = \mathbf{)}$. Otherwise r is *consistent*.

For example, $r \sqsubseteq (1.2, 1.2]$ and $v \sqsubseteq [4.56, 2.33)$ are inconsistent in the real domain, $i \sqsubseteq [4, 0)$ is inconsistent in the integer domain and $s \in [\{1\}, \{3\}]$ is inconsistent in the domain *Set Integer* with the inclusion as ordering relation.

Constraint Propagation. Constraint propagation is executed from non-simple constraints by evaluating of indexical terms with respect to simple constraints.

For example, let $c_y = y \sqsubseteq (8.90, 78.91]$ be a simple constraint, and consider the definition of $+_l$ and $+_r$ of Table 1 with $L = L' = \Re$. Then, the constraint

$c_z = z \sqsubseteq min(y) +_l \mathbf{(1.00}, max(y) +_r \mathbf{4.00]}$

is propagated to constraint

$c'_z = z \sqsubseteq \mathbf{(9.90, 82.91]}$

since $eval_{c_y}(min(y)) = \mathbf{(8.90}$ and $eval_{c_y}(max(y)) = \mathbf{78.91]}$; then $\mathbf{(8.90} \ +_l \ \mathbf{(1.00} = \mathbf{(9.90}$ and $\mathbf{78.91]} \ +_r \ \mathbf{4.00]} = \mathbf{82.91]}$

We write $c_z \leadsto^{c_y} c'_z$ to say that c_z is propagated to c'_z via c_y.

Propagation and narrowing of sets of constraints. A *constraint store* S is a set of interval constraints. If S just contains simple constraints then S is said to be simple. Let X be the set of variables constrained in a store S by an interval constraint. Then, S is *stable* if it contains at most one simple interval constraint defined for each $x \in X$. For example, the constraint store $\{x \sqsubseteq [1, 5), r \sqsubseteq [3.45, 6.78]\}$ is stable and simple.

Let S be a simple stable constraint store and C a set of interval constraints. Then C *is propagated to* the simple constraint store C' *using* S, denoted by $C \leadsto^S C'$, if $C' = \{c' \mid \exists c \in C \wedge c'' \in S \,.\, c \leadsto^{c''} c'\}$.

If S is a simple constraint store, and X the set of variables constrained in S, then we write $S \mapsto S'$ to indicate that $S' = \bigcup_{x \in X} \ \cap_L \{x \sqsubseteq r \mid x \sqsubseteq r \in S\}$. Note that S' is stable and simple.

Example. Let $X = \{x, y, i\}$ where $x, y \in V_\Re$ and $i \in V_{Integer}$, and let

$$\begin{aligned} S = \{ \ & x \sqsubseteq [1.0, 15.0], x \sqsubseteq (8.3, 20.4], \\ & y \sqsubseteq [1.2, 10.5), y \sqsubseteq (5.6, 15.3), \\ & i \sqsubseteq [0, 10], i \sqsubseteq [2, 15) \ \} \end{aligned}$$

be a simple constraint store. Then, $S \mapsto S'$ where

$$S' = \{ \ x \sqsubseteq (8.3, 15.0], y \sqsubseteq (5.6, 10.5), i \sqsubseteq [2, 10] \ \}.$$

[3] Refer to [9] for more information about the evaluation function.

Moreover, by considering the definition of $trunc_l$ and $trunc_r$ in Table 1, we have

$$\{\ x \sqsubseteq min(y), \mathbf{20.4}],\ i \sqsubseteq trunc_l(min(y)), trunc_r(max(y))\ \} \\ \leadsto^{S'} \{\ x \sqsubseteq (5.6, 20.4],\ i \sqsubseteq (5, 10)\ \}.$$

Note that constraint $i \sqsubseteq trunc_l(min(y)), trunc_r(max(y))$ propagates values from the real domain to the integer domain.

Operational semantics. A schema for the operational semantics was first described in [7]. In [6, 9] we have modified such a schema and provided the basic methodology. We also proved some properties of the main algorithm such as termination and correctness. This schema is valid, with minor modifications for our generalisation to multiple domains.

The whole formal theory and non-trivial examples are available at [9].

4.4 Monotonicity of constraints.

Our framework imposes the monotonicity law for operators. This imposition is provided in order to avoid the use of non-simple constraints that produce no constraint propagation. For example, consider the operator $-_r$, defined in Table 1, with $L = L' = Integer$ and the usual definition of the operator $-$ for integers. The constraint $c_x = x \sqsubseteq [\mathbf{10}, \mathbf{20}] -_r max(y)$ is non-monotonic since as the evaluation of $max(y)$ decreases (resp. increases) in $Integer_r$, $\mathbf{20}] -_r max(y)$ increases (resp. decreases) in $Integer_r$ so that the range associated to x increases. For instance, $[0, 5] <_{R_{Integer}} [0, 10]$ and let $c_1 = y \sqsubseteq [0, 10]$ and $c_2 = y \sqsubseteq [0, 5]$. Then $x \sqsubseteq [10, 20] -_r max(y) \leadsto^{c_1} x \sqsubseteq [10, 10]$ and $x \sqsubseteq [10, 20] -_r max(y) \leadsto^{c_2} x \sqsubseteq [10, 15]$ but $[10, 10] <_{R_{Integer}} [10, 15]$.

Observe that a range such as $[10, 20] -_r max(y)$ could not contribute to constraint propagation. To satisfy the monotonicity law for operators, the binary operator $-_r$ has to be declared as $-_r :: L_r \times L_l \to L_r$. Therefore, since $max(y) \notin Integer_l$ a constraint such as $x \sqsubseteq [\mathbf{10}, \mathbf{20}] -_r max(y)$ is not allowed.

5 High level generic constraints

Using the primitive constraint $x \sqsubseteq r$ it is possible to define high level constraints, in the style of clp(FD) [4] although, unlike clp(FD), our framework also allows the definition of both generic and overloaded constraints. A constraint is *generic* in $\mathcal{L}$ if its definition is independent of the computation domain and it can be used on all $L \in \mathcal{L}$. Consider a subset $\mathcal{L}_1 \subset \mathcal{L}$ with $card(\mathcal{L}_1) > 1$. A constraint is *overloaded* in $\mathcal{L}$ if it can only be used on all $L \in \mathcal{L}_1$ but not on all $L \in \mathcal{L}$. For example, the definition of the 'less-or-equal-than' constraint shown above is generic for any $x, y \in V_L$ since each $L \in \mathcal{L}$ contains both (lifted) top and bottom elements.

$$x \leq y \ \Leftrightarrow x \sqsubseteq [\bot_L, max(y), \\ y \sqsubseteq min(x), \top_L]$$

For instance, consider $\mathcal{L} = \{Integer, \Re, Bool\}$ with the usual ordering in the integer and real domain and $false < true$ in $Bool$, the variables $x, y \in V_{Integer}$, $z, w \in V_{\Re}$, $b_1, b_2 \in V_{Bool}$ and the constraint store

$$\begin{aligned} S = \{ \ & x \sqsubseteq [0,23), \ y \sqsubseteq [5,15], \\ & z \sqsubseteq (2.0, 10.5), \ w \sqsubseteq [0.0, 12.0], \\ & b_1 \sqsubseteq [false, true], \ b_2 \sqsubseteq [false, false] \ \}. \end{aligned}$$

Then,$\{x \leq y\} \rightsquigarrow^S S_1$, $\{z \leq w\} \rightsquigarrow^S S_2$ and $\{b_1 \leq b_2\} \rightsquigarrow^S S_3$ where $S_1 = \{x \sqsubseteq [\bot_{Integer}, 15], y \sqsubseteq [0, \top_{integer}]\}$, $S_2 = \{z \sqsubseteq [\bot_{\Re}, 12.0], w \sqsubseteq (2.0, \top_{\Re}]\}$ and $S_3 = \{b_1 \sqsubseteq [false, false], b_2 \sqsubseteq [false, true]\}$. Also, $S \cup S_1 \cup S_2 \cup S_3 \mapsto \{x \sqsubseteq [0,15], y \sqsubseteq [5,15], z \sqsubseteq (2.0, 10.5), w \sqsubseteq (2.0, 12.0], b_1 \sqsubseteq [false, false], b_2 \sqsubseteq [false, false]\}$.

6 Propagation on direct combinations: an example

Consider the following example taken from [12] in the domain of planar geometry.

Problem Statement. Find all pairs of non trivial quadrilaterals satisfying the following set of constraints:

1. C_1: All vertices have integer coordinates in $\{1..n\}$.
2. C_2: All quadrilaterals are straight rectangles.
3. C_3 : The two rectangles do not intersect.

Representation of the problem. Given the plane $(0,0) \times (n,n)$ a rectangle r is identified by its lower left corner and its upper right corner (for simplicity, the set of constraints $\{x \leq y, y \sqsubseteq [l,l]\}$ where $x, y \in V_L$ and $l \in L$ will just be denoted as $x \leq l$. And analogously for $l \leq x$).

In the following we show how it is possible to solve this problem on different domains by using the generic constraint $\leq /2$ defined in Section 5.

Designing the problem in the integer domain. Let $integer^+$ be the (lifted) lattice of the positive integers plus value 0 where $\bot_{integer^+} = 0$ and $\top_{integer^+}$ is a ficticial bound. A first formulation consists of identifying a rectangle by the coordinates of its corners and consider them as atomic values in the $integer^+$ domain. Constraints are stated entirely in terms of the $integer^+$ domain.

Let (p_{1x}, p_{1y}),(p_{2x}, p_{2y}) and (p_{3x}, p_{3y}),(p_{4x}, p_{4y}) be the coordinates of the two corners identifying a rectangle r_1 and r_2 respectively (see diagram (a) in Figure 1). Constraint C_1 can be stated as follows:

$$\begin{aligned} &(1 \leq p_{1x}) \wedge (p_{1x} \leq n) \wedge (1 \leq p_{1y}) \wedge (p_{1y} \leq n) \wedge \\ &(1 \leq p_{2x}) \wedge (p_{2x} \leq n) \wedge (1 \leq p_{2y}) \wedge (p_{2y} \leq n) \wedge \\ &(1 \leq p_{3x}) \wedge (p_{3x} \leq n) \wedge (1 \leq p_{3y}) \wedge (p_{3y} \leq n) \wedge \\ &(1 \leq p_{4x}) \wedge (p_{4x} \leq n) \wedge (1 \leq p_{4y}) \wedge (p_{4y} \leq n) \wedge \end{aligned}$$

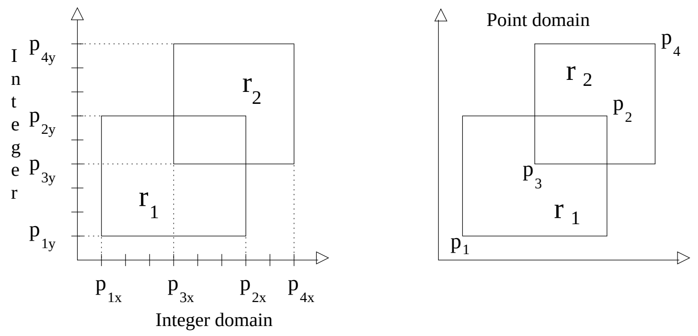
C_2 as follows:

$$(p_{1x} < p_{2x}) \wedge (p_{1y} < p_{2y}) \wedge (p_{3x} < p_{4x}) \wedge (p_{3y} < p_{4y})$$

where constraint $</2$ is just defined as $x < y \Leftrightarrow x \leq y \wedge x \neq y$. And C_3 as:

$$(p_{2x} < p_{3x}) \vee (p_{2y} < p_{3y})$$

Designing the problem in the *Point* domain. Our lattice-based framework allows for new computation domains to be constructed from previously defined domains. The *Point* domain can be constructed from the direct product $\langle integer^+, integer^+\rangle$. Note that $\perp_{Point} = \langle 0, 0\rangle$ and $\top_{Point} = \langle \top_{integer^+}, \top_{integer^+}\rangle$. Then, r_1 and r_2 can be identified by the points p_1,p_2 and p_3,p_4 respectively (see



(a) Representation in the Integer domain

(b) Representation in the Point domain

Figure1. The problem of non-intersecting rectangles

diagram (b) in Figure 1) and constraints C_1, C_2 and C_3 can be stated directly in the *Point* domain. Constraint C_1 is stated as follows:

$$(\langle 1, 1\rangle \leq p_1) \wedge (p_1 \leq \langle n, n\rangle) \wedge (\langle 1, 1\rangle \leq p_2) \wedge (p_2 \leq \langle n, n\rangle) \wedge$$
$$(\langle 1, 1\rangle \leq p_3) \wedge (p_3 \leq \langle n, n\rangle) \wedge (\langle 1, 1\rangle \leq p_4) \wedge (p_4 \leq \langle n, n\rangle)$$

constraint C_2 as $(p_1 < p_2) \wedge (p_3 < p_4)$ and constraint C_3 should be stated as $p_3 \not\leq p_2$. However, it is not clear how to express this last constraint since the meaning is not exactly $p_2 < p_3$ (note that there are values in the point domain such that $p_2 \not< p_3$). In the following we propose a solution based on a novel concept of solver cooperation that we call *higher-order constraint propagation.*

Higher-order constraint propagation. As it was shown above, constraint C_3 can easily be defined in the integer+ domain. With the generalisation of operators shown in preceding sections, it is also easy propagate constraints from the $integer+$ domain to the $Point$ domain. Consider the following constraints

$$p_2 \sqsubseteq min(p_{2x}) \ \diamond_l \ min(p_{2y}), max(p_{2x}) \ \diamond_r \ max(p_{2y})$$
$$p_3 \sqsubseteq min(p_{3x}) \ \diamond_l \ min(p_{3y}), max(p_{3x}) \ \diamond_r \ max(p_{3y}) \qquad (1)$$

For all $L \in \mathcal{L}$, $\diamond_l$ and $\diamond_r$ are generically defined in Table 1 where we consider $L' = \langle L, L \rangle$. For example, in the integer domain $\mathbf{3}] \diamond_r \mathbf{2)} = \langle 3, 2 \rangle)$ and $[\mathbf{3} \diamond_l \mathbf{(2} = (\langle 3, 2 \rangle$. Also in the $Point$ domain $\langle 1, 3 \rangle] \diamond_r \langle 4, 4 \rangle] = \langle \langle 1, 3 \rangle, \langle 4, 4 \rangle \rangle]$. Then, from constraints in (1), any change in the coordinates $p_{2x}, p_{2y}, p_{3x}, p_{3y}$ is directly propagated, by evaluation of the indexical terms, to points p_1 and p_2.

Constraint propagation can also affect directly to rectangles r_1 and r_2. Let now $Rectangle$ be the direct product $\langle Point, Point \rangle$. $Rectangle$ is a lattice. Note that $\bot_{Rectangle} = \langle \langle 0, 0 \rangle, \langle 0, 0 \rangle \rangle$ and $\top_{Rectangle} = \langle \top_{Point}, \top_{Point} \rangle$. Then values corresponding to the $Point$ domain can be directly propagated to the $Rectangle$ domain by introducing the following constraints:

$$r_1 \sqsubseteq min(p_1) \ \diamond_l \ min(p_2), max(p_1) \diamond_l max(p_2)$$
$$r_2 \sqsubseteq min(p_3) \ \diamond_l \ min(p_4), max(p_3) \diamond_l max(p_4)$$

where $r_1, r_2 \in V_{Rectangle}$ and $p_1, p_2, p_3, p_4 \in V_{Point}$.

7 Constraint solving on linear combinations

In this section we show that our extension is useful not only on the direct or lexicographic combination of domains but also on more interesting combinations such as the linear sum of domains.

7.1 An overloaded *plus* constraint

Consider the definition of the operators $-_l$, $-_r$, $+_r$ and $+_l$ shown in Table 1 and the following definition of a *plus/3* constraint:

$$\begin{aligned} plus(x, y, z) \Leftrightarrow x \ &\sqsubseteq min(z) -_l max(y), max(z) -_r min(y), \\ y \ &\sqsubseteq min(z) -_l max(x), max(z) -_r min(x), \\ z \ &\sqsubseteq min(x) +_l min(y), max(x) +_r max(y). \end{aligned}$$

where $x, y \in V_L$ and $z \in V_{L'}$. This constraint is overloaded since it is valid for any domain in which operators $+$ and $-$ are defined so that if $a \in L'$ and $b, c \in L$, then $b + c \in L'$ and $a - b \in L$.

For example, consider that $L = L'$ with $L \in \{Integer, \Re\}$, the usual definitions for $+$ and $-$ in L, the variables $x, y, z \in V_{Integer}$, the variables $r, w, t \in V_{\Re}$ and the following constraint store:

$$S = \{t \sqsubseteq [1.0, 4.0], w \sqsubseteq (0.0, 90.0], x \sqsubseteq [1, 2], y \sqsubseteq [2, 9)\}$$

Then, $plus/3$ executes constraint propagation on these domains. For example,

$$\{plus(r, w, t)\} \leadsto^S \{ \ r \sqsubseteq [-89.0, 4.0) \ \} \text{ and } \{plus(x, y, z)\} \leadsto^S \{ \ z \sqsubseteq [3, 11) \ \}.$$

7.2 A more motivating example

In current CLP systems, most of the arithmetical operations executed on a particular domain return a value belonging to such a domain. For example, an expression such as $a + b$, where a, b are reals, returns a real value. However, in practice there are well-known domains such as the binary and hexadecimal domains in which this is not necessarily true (see picture 2). For example, in the hexadecimal domain, 'F'+'F' does not belong to the hexadecimal domain but to the lexicographic product $(hexadecimal, hexadecimal)$. This means that the standard definition of the operator $+$ is not valid.

$$\left.\begin{array}{r}1\\+1\\\hline 10\end{array}\quad\begin{array}{r}1\\+0\\\hline 01\end{array}\quad\begin{array}{r}0\\+1\\\hline 01\end{array}\quad\begin{array}{r}0\\+0\\\hline 00\end{array}\right\}\begin{array}{l}\text{binary}\\ \\ \text{binary x binary}\end{array}\qquad\left.\begin{array}{r}F\\+F\\\hline 1E\end{array}\quad\begin{array}{r}F\\+E\\\hline 1D\end{array}\;\cdots\cdots\;\begin{array}{r}0\\+0\\\hline 00\end{array}\right\}\begin{array}{l}\text{hexadecimal}\\ \\ \text{hexadecimal x hexadecimal}\end{array}$$

Figure2. Linear combinations of the sum of domains

A naive solution would extend the operation $+$ with an extra *carry* argument. For example, 'F'+'F'=('1','E'). We propose a more expressive and elegant solution consisting of propagating values from basic domains to their combinations.

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

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

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

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

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

It is routine to check that the linear sum of lattices is a lattice.

Now 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'. Then the lattice of hexadecimal digits, denoted by $hexadecimal$, is defined as the lattice $0to9 \oplus AtoF$.

We overload the definitions of operators $+_l, +_l, -_r$ and $-_l$ shown in Table 1 by considering $L = hexadecimal$ and $L' = (L, L)$ (that is, the lexicographic product (L, L)). Their definition depends on the operator $+$ and the operator $-$ that are done as usual, that is, '0'+'0'=('0','0'), ..., '1'+ 'F'=('1','0'), ..., 'F'+'F'=('1','E') and ('1','E')-'F'='F', ... ,('1','0')-'1'='F', ... ,etc. Consider the following set of constraints:

$$S = \{ \ h_1 \sqsubseteq [\text{'0','F'}], h_2 \sqsubseteq (\text{'A','F'}] \ \}$$

where $h_1, h_2 \in V_L$. Then, $\{plus(h_1, h_2, h_3)\} \leadsto^S \{ \ h_3 \sqsubseteq [(\text{'0','A'}),(\text{'1','E'})] \ \}$ where $h_3 \in V_{L'}$. Note that the constraint $plus/3$, defined in Section 7.1, is also valid in the hexadecimal domain.

To be more consistent with the definition of the operators $+$ and $-$, we also should include a constraint such as $h_3 \leq$ ('1','E') (see definition of constraint $\leq/2$ in Section 5) since the maximum of the sum of two single hexadecimals is ('1','E'). Alternatively, we can declare constraints for higher-order constraint solving using the higher-order approach shown in Section 6. For example, consider the constraint

$$c \ = \ h_3 \sqsubseteq min(h_4) \oslash_l [\textbf{'0'}, max(h_4) \oslash_r \textbf{'E'}]$$

where $h_4 \in V_L$, and $\oslash_l/2$ and $\oslash_r/2$ are generic operators defined analogously to the operators $\diamond_l$ and $\diamond_r$ (see Table 1) but where $L' = (L, L)$ (that is, the lexicographic product). By adding the constraint $c' = h_4 \sqsubseteq [\text{'0','1'}]$, h_3 is restricted to the range $[(\text{'0','0'}),(\text{'1','E'})]$. Note that $c \leadsto^{c'} h_3 \sqsubseteq [(\text{'0','0'}),(\text{'1','E'})]$.

8 Even more expressivity!

So far, we have shown that our generalisation allows constraint propagation from basic domains to combined domains and vice versa. In this section we show that this generalisation also enables information to flow between non-related domains. For example, consider the following conditional sentence:

$$\textbf{if } b \textbf{ then } i = trunc(r * r) \textbf{ else } i = trunc(r)$$

where b is a Boolean, r a positive real and i a positive integer. Then, the meaning of this sentence can be captured by the following interval constraint:

$$c_1 \ = \ i \sqsubseteq min(b) \rhd_l min(r), max(b) \rhd_r max(r)$$

where $r \in V_{\Re+}$, $b \in V_{Bool}$, $i \in V_{integer^+}$ and $\rhd_l/2$ and $\rhd_r/2$ are binary operators defined and declared in Table 1. Constraint propagation is executed as usual. For example, consider the store $S = \{r \sqsubseteq [2.3, 9.5), b \sqsubseteq [false, true]\}$, then[4] $\{c_1\} \leadsto^S \{ \ i \sqsubseteq [2, 90) \ \}$. More propagation may be obtained by adding other constraints and by constraint narrowing. For example, $S \cup \{b \sqsubseteq [true, true]\} \mapsto S_1$ with $S_1 = \{r \sqsubseteq [2.3, 9.5), b \sqsubseteq [true, true] \ \}$. Then[5] $\{c_1\} \leadsto^{S_1} \{ \ i \sqsubseteq [5, 90) \ \}$.

This simple example illustrates how the generalised framework provides the mechanism for relating different computation domains and propagating the constraints between domains.

[4] Note that $i \sqsubseteq [2, 90)$ comes from $i \sqsubseteq [false \rhd_l [2.3, true] \rhd_r 9.5)$.
[5] Note that $i \sqsubseteq [5, 90)$ comes from $i \sqsubseteq [true \rhd_l [2.3, true] \rhd_r 9.5)$

9 Conclusions and further work

In this paper we have generalised our work in [7] to allow for multiple domains. First, we have developed the theoretical work and shown that a unique solver based on a primitive constraint to the style of clp(FD) can support (system or user defined) domains with lattice structure.

Then, we have shown that our framework enables a novel constraint propagation mechanism that allows the information flow between the multiple domains. This mechanism means a powerful way in which solvers of different domains can cooperate. Specially interesting is the propagation of constraints on combined domains. These domains can be easily constructed in our framework since the only requirement for our framework is that the computational domain must be a lattice and therefore new domains can be obtained from previously defined domains using standard combinators (such as direct product and sum). We have provided examples to highlight the potential and expressivity of the cooperation of solvers with non-trivial examples.

We have also shown how generic and overloaded constraints are easily defined in our framework and how to control the monotonicity of constraints by imposing the monotonicity of operators.

The cooperation of solvers comes from the operator definitions and not from the set of constrained variables. So far, we have used sets of constrained variables associated to each computation. It seems interesting and promising to study the idea of considering sub-lattices as new computation domains. Sub-lattices could provide a means of having variable sets of a sublattice being also allowed as variables in the main lattice. This needs careful consideration and we think that it is worth to consider in the future.

To demonstrate the feasibility of our ideas we have developed a CLP language [9] and built a prototype implementation so that all the examples shown in this paper has been tested in this prototype. This is continuously being improved and extended but the latest version may be obtained in [5]. The prototype is built with CHR's [10] and it is well known that although expressive, there are other more efficient systems [8]. This is the main reason because we have not compared our prototype with other systems. We are working on the developing of a more efficient implementation and plan to compare it with other systems in the future.

References

1. F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 1–21. Springer Verlag, 1994.
2. F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and Boolean constraints. *The Journal of Logic Programming*, 32(1):1–24, July 1997.
3. J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.
4. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *The Journal of Logic Programming*, 27(3):185–226, 1996.

5. A.J. Fernández. clp(L) version 0.21, user manual. Available from *http : //www.lcc.uma.es/~afdez/generic*, 1999.
6. A.J. Fernández and P.M. Hill. Interval constraint solving over lattices using chaotic iterations. In K.R. Apt, C. Kakas, E. Monfroy, and F. Rossi, editors, *ERCIM Workshop on Constraints*, Cyprus, 1999.
7. A.J. Fernández and P.M. Hill. An interval lattice-based constraint solving framework for lattices. In Aart Middeldorp and Taisuke Sato, editors, *4th International Symposium on Functional and Logic Programming (FLOPS'99)*, number 1722 in LNCS, pages 194–208, Tsukuba, Japan, November 1999. Springer Verlag.
8. A.J. Fernández and P.M. Hill. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Constraints*, 5(3):275–301, 2000.
9. A.J. Fernández and P.M. Hill. Interval-lattice based constraint logic programming on lattice structure domains. Unpublished draft. Available in *http : //www.lcc.uma.es/~afdez/unpublished*, February 2000.
10. T. Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37:95–138, 1998.
11. J.H.M. Lee and Van Emdem. Interval computation as deduction in CHIP. *The Journal of Logic Programming, Special Issue:Constraint Logic Programming*, 16(3-4), 1993.
12. F. Pachet and P. Roy. Integrating constraint satisfaction techniques with complex object structures. In *Annual Conference on the British Computer Society on Expert Systems (ES'95)*, pages 11–22, 1995. Cambridge.

Table1. Definition of operators

$trunc_l :: \Re_l \rightarrow Integer_l$ $trunc_l(\{\mathbf{a}) = \{\ trunc(\mathbf{a})$	$trunc_r :: \Re_r \rightarrow Integer_r$ $trunc_r(\mathbf{a}\}) = trunc(\mathbf{a})\ \}$
$+_l :: L_l \times L_l \rightarrow L'_l$ $\{_1\mathbf{a}\ +_l\ \{_2\mathbf{b}\ =\ min_B(\{_1, \{_2)\ a + b$	$+_r :: L_r \times L_r \rightarrow L'_r$ $\mathbf{a}\}_1\ +_r\ \mathbf{b}\}_2\ =\ a + b\ min_B(\}_1, \}_2)$
$-_l :: L'_l \times L_r \rightarrow L_l$ $\{_1\mathbf{a}\ -_l\ \mathbf{b}\}_2\ =\ min'_B(\{_1, \{_2)\ a - b$	$-_r :: L'_r \times L_l \rightarrow L_r$ $\mathbf{a}\}_1\ -_r\ \{_2\mathbf{b}\ =\ a - b\ min'_B(\}_1, \}_2)$
$\diamond_l :: L_l \times L_l \rightarrow L'_l$ $\{_1\mathbf{a} \diamond_l \{_2\mathbf{b} = min_B(\{_1, \{_2)\ \langle a, b\rangle$	$\diamond_r :: L_r \times L_r \rightarrow L'_r$ $\mathbf{a}\}_1 \diamond_r \mathbf{b}\}_2 = \langle a, b\rangle\ min_B(\}_1, \}_2)$
$\triangleright_l :: Boolean_l \times \Re_l \rightarrow Integer_l$ $\{_1\mathbf{true} \triangleright_l \{_2\mathbf{r} = min_B(\{_1, \{_2)\ trunc(r * r)$ $\{_1\mathbf{false} \triangleright_l \{_2\mathbf{r} = min_B(\{_1, \{_2)\ trunc(r)$	$\triangleright_r :: Boolean_r \times \Re_r \rightarrow Integer_r$ $\mathbf{true}\}_1 \triangleright_r \mathbf{r}\}_2 = trunc(r * r)\ min_B(\}_1, \}_2)$ $\mathbf{false}\}_1 \triangleright_r \mathbf{r}\}_2 = trunc(r)\ min_B(\}_1, \}_2)$