Compiling Business Rules in a Geometric Constraint over k-Dimensional Objects and Shapes

Mats Carlsson
SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

Nicolas Beldiceanu
École des Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France
Nicolas.Beldiceanu@emn.fr

Julien Martin
INRIA Rocquencourt, BP 105, FR-78153 Le Chesnay Cedex, France
Julien.Martin@inria.fr

SICS Technical Report T2009:02
ISSN: 1100-3154
ISRN: SICS-T–2009/02-SE

Abstract: It is well known that real-life applications rarely admit a constraint model expressed purely in terms of a few global constraints. Usually, the global constraints capture a relaxed form of the problem, but needs additional side-constraints to capture the full problem. Handling such side-constraints inside the global constraints, as opposed to in conjunction with it, improves propagation. Historically, this has been done by extending the global constraints with a host of specific options, each connected to a specific filtering method. Being able to express and filter side-constraints in a more uniform and systematic way would seem a more elegant and manageable solution.

This report presents such a mechanism for the global non-overlapping constraint *geost*, which handles the location in space of k-dimensional objects. Side-constraints are expressed as rules written in a language based on arithmetic and first-order logic, which should hold among the objects. We explain in detail the way the rules are compiled into a form that is accessed by the constraint's sweep-based filtering algorithm. In a first step, the rules are rewritten to Quantifier-Free Presburger Arithmetic (QFPA) formulas. Secondly, such formulas are transformed to generators of k-dimensional forbidden sets. Thirdly, the generators are transformed to procedures answering queries about candidate coordinate points for a given object. Such queries are at the heart of the filtering algorithm.

The business rules allow to express a great variety of packing and placement constraints, while admitting effective filtering of the domain variables of the k-dimensional object, without the need to use spatial data structures. The constraint was used to directly encode the packing knowledge of a major car manufacturer, and was evaluated on several benchmarks.

Keywords: Global Constraint; Geometric Constraint; Rule Language; Sweep; Quantifier-Free Presburger Arithmetic.

Contents

1 Introduction

It is well known that real-life applications rarely admit a constraint model expressed purely in terms of a few global constraints. Usually, the global constraints capture a relaxed form of the problem, but needs additional side-constraints to capture the full problem. Handling such side-constraints directly inside the global constraints, as opposed to in conjunction with it, provides opportunities to improve propagation.

To express the side-constraints, global constraints have traditionally been extended with extra options or arguments. For example, the *diffn* constraint [1] of CHIP provides, beside non-overlapping, a variety of other geometrical constraints (in fact more than 10 side constraints). This was also the case for the *cycle* and *tree* constraints [1, 2] where, beside a graph partitioning constraint, a variety of useful side constraints were also provided. Even if this makes sense when one wants to efficiently solve specific real-life applications, this proliferation of arguments and options has two major drawbacks:

- Having a lot of ad-hoc side constraints is too specific and can sometimes be quite frustrating since it does not allow to express a small variant of an existing side constraint.
- Designing a filtering algorithm for each side constraint independently is not enough and managing the interaction of several side constraints becomes more and more challenging as the number and variety of side constraints increase [2, 3].

Instead, we have taken the approach of defining a language based on arithmetic and first-order logic, which should hold among the objects. The language can also be seen as a natural target constraint of the `Rules2CP` modeling language [4]. Side-constraints are expressed as a set of rules written in the language. This approach addresses the two issues mentioned above in the following way:

1. Having a rule language for expressing side constraints is obviously more flexible than having a large set of predefined side constraints.
2. As we will see later on, our filtering algorithm allows to some extent to take into account the interaction among all rules.
3. The *geost* constraint has a greedy assignment mode [5], which attempts to fix all variables in one go, ignoring any constraint that is in conjunction with *geost*. The rule language makes it possible to capture many packing problems in a single *geost* constraint, in which case this mechanism can lead to significant speedups.

This report presents the language and its compilation to a form that is accessed by the constraint's sweep-based filtering algorithm. In a first step, the rules are rewritten to Quantifier-Free Presburger Arithmetic (QFPA) formulas. Secondly, such formulas are transformed to generators of k-dimensional forbidden sets. Thirdly, the generators are transformed to procedures answering queries about candidate coordinate points for a given object. Such queries are at the heart of the filtering algorithm. The report extends our conference paper [6] in the following way:

1. In [6], we only had a proof-of-concept Prolog implementation of the filtering algorithm. In this report, we report results from integrating the rules into the sweep-based filtering algorithm of *geost*, which is written in C [7]. Consequently, we can use the rules together with the dedicated filtering methods for non-overlapping [8] and symmetry breaking [5]: the sweep-point kernel aggregates information coming from the rules with information coming from the built-in methods.
2. The performance evaluation was extended with several new benchmarks [9, 10] involving side-constraints.

The context of the work is the global constraint $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ for handling the location in space of k-dimensional objects, $\mathcal{O}$ ($k \in \mathbb{N}^+$), each of which taking a shape among a set of shapes $\mathcal{S}$, subject to a set of rules given in the $\mathcal{R}$ argument. Each shape from $\mathcal{S}$ is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k-dimensional space at the given offset with the given sizes. More precisely a ***shifted box*** $s \in \mathcal{S}$ is an entity defined by its shape id $s.sid$, shift offset $s.t[d]$, $1 \leq d \leq k$, and

sizes $s.l[d]$ (where $s.l[d] > 0$ and $1 \leq d \leq k$). All attributes of a shifted box are integer values. A *shape* is a collection of shifted boxes all sharing the same shape id. Each object $o \in \mathcal{O}$ is an entity defined by its unique object id $o.oid$ (an integer), shape id $o.sid$ (an integer if the object has a fixed shape, or a domain variable for *polymorphic* objects, which have alternative shapes), and origin coordinate $o.x[d]$, $1 \leq d \leq k$ (integers, or domain variables that do not occur anywhere else in the constraint).[1] Objects and shifted boxes may also have additional, integer (but see also Section 7) attributes, such as weight, customer, or fragility, used by the rules.

We denote by *forbidden set for a logical formula r and an object o* a finite subset $S \subset \mathbb{Z}^k$ such that if the origin of o is in S, r is disentailed. Such a forbidden set can also be seen as the k-dimensional generalization of a set of inconsistent assignments [11]. A *forbidden orthotope* f is a forbidden set f shaped as an orthotope (hyper-rectangle). We represent f as a pair $(f.\min, f.\max)$ of a lower bound vector $f.\min$ and an upper bound vector $f.\max$.

Each rule in $\mathcal{R}$ is a first-order logical formula over the attributes of objects and shifted boxes. Forbidden sets are automatically derived from such formulas and used by the sweep-based algorithm of *geost*, which recursively traverses the placement space for each coordinate in increasing as well as decreasing lexicographic order and skips unfeasible points that are located inside such forbidden sets. This contrasts with the previous version of *geost*, where an ad-hoc algorithm computing the multi-dimensional forbidden sets had to be worked out for each side-constraint. $\mathcal{R}$ may also contain macros, providing abbreviations for expressions occurring in formulas or in other macros. All but a core fragment of the language can be eliminated by equivalence-preserving rewriting. The remaining fragment is a subset of Quantifier-Free Presburger Arithmetic (QFPA), which has a very simple semantics and, as we will show, is amenable to efficient compilation. Constraint satisfaction problems using quantified formulas (QCSP) have for instance been studied by Benedetti et al. [12], mostly in the context of modeling games. QCSP does not provide disjunction but actively uses quantifiers in the evaluation, whereas we eliminate all quantifiers in the process of rewriting to QFPA.

Example 1 *This running example will be used to illustrate the way we compile rules to code used by the sweep-based algorithm [7] for filtering the coordinates of each object. Suppose that we have five objects o_1, o_2, o_3, o_4 and o_5 such that:*

- *o_1, o_2 and o_4 are rectangles fixed at $(1, 2)$, $(3, 3)$ and $(3, 7)$ of respective size 3×1, 1×1 and 3×1.*
- *The rectangle o_3 is fixed at $(2, 5)$ but not its shape variable s_3, which can take values corresponding to size 1×2 or 2×1. We will denote by ℓ_{31} resp. ℓ_{32} the length resp. height of o_3.*
- *The coordinates of the non-fixed square o_5 of size 2×2 correspond to the two variables $x_{51} \in [1, 9]$ and $x_{52} \in [1, 6]$.*
- *o_2, o_4 and o_5 have the additional attribute $type$ with value 1 whereas o_1 and o_3 have $type$ with value 2.*
- *Two rules must be obeyed; see Fig. 1:*
  - *All objects should be mutually non-overlapping.*
  - *If the $type$ attribute of two objects both equal 1, the two objects should not touch, not even their corners.*

Fig. 1 shows one solution that satisfies the above constraints.

□

[1] A *domain variable* v is a variable ranging over a finite set of integers denoted by $\mathrm{dom}(v)$; $\underline{v}$ and $\overline{v}$ denote respectively the minimum and maximum possible values for v.

$$\mathcal{O} = \begin{bmatrix} \text{object}(1,1,[1,2],[\text{type-2}]), \\ \text{object}(2,2,[3,3],[\text{type-1}]), \\ \text{object}(3,s_3,[2,5],[\text{type-2}]), \\ \text{object}(4,1,[3,7],[\text{type-1}]), \\ \text{object}(5,5,[x_{51},x_{52}],[\text{type-1}]) \end{bmatrix} \qquad \mathcal{S} = \begin{bmatrix} \text{sbox}(1,[0,0],[3,1]), \\ \text{sbox}(2,[0,0],[1,1]), \\ \text{sbox}(31,[0,0],[1,2]), \\ \text{sbox}(32,[0,0],[2,1]), \\ \text{sbox}(5,[0,0],[2,2]) \end{bmatrix}$$

$\mathcal{R} =$

$ori(o,s,d) \Longrightarrow o.x[d] + s.t[d]$
$end(o,s,d) \Longrightarrow o.x[d] + s.t[d] + s.l[d]$
$overlap(D,o_i,s_i,o_j,s_j) \Longrightarrow$
$\quad \forall d \in D:\ end(o_i,s_i,d) > ori(o_j,s_j,d) \wedge end(o_j,s_j,d) > ori(o_i,s_i,d)$
$meet(D,o_i,s_i,o_j,s_j) \Longrightarrow$
$\quad (\forall d \in D:\ end(o_i,s_i,d) \geq ori(o_j,s_j,d) \wedge end(o_j,s_j,d) \geq ori(o_i,s_i,d)) \wedge$
$\quad (\exists d \in D:\ end(o_i,s_i,d) = ori(o_j,s_j,d) \vee end(o_j,s_j,d) = ori(o_i,s_i,d))$
$all\_not\_overlap(D, OIDs) \Longrightarrow$
$\quad \forall o_i \in OIDs, \forall s_i \in o_i.sid\,, \forall o_j \in OIDs:$
$\quad\quad o_i.oid < o_j.oid \Rightarrow (\forall s_j \in o_j.sid:\ \neg overlap(D,o_i,s_i,o_j,s_j))$
$all\_type1\_not\_meet(D, OIDs) \Longrightarrow$
$\quad \forall o_i \in OIDs, \forall s_i \in o_i.sid\,, \forall o_j \in OIDs:$
$\quad\quad o_i.oid < o_j.oid \wedge o_i.type = 1 \wedge o_j.type = 1 \Rightarrow$
$\quad\quad\quad \forall s_j \in o_j.sid:\ \neg meet(D,o_i,s_i,o_j,s_j)$
$all\_not\_overlap\_sboxes([1,2],[1,2,3,4,5])$
$all\_type1\_not\_meet\_sboxes([1,2],[1,2,3,4,5])$

$s_3 \in [31,32], x_{51} \in [1,9], x_{52} \in [1,6], geost(2,\mathcal{O},\mathcal{S},\mathcal{R})$

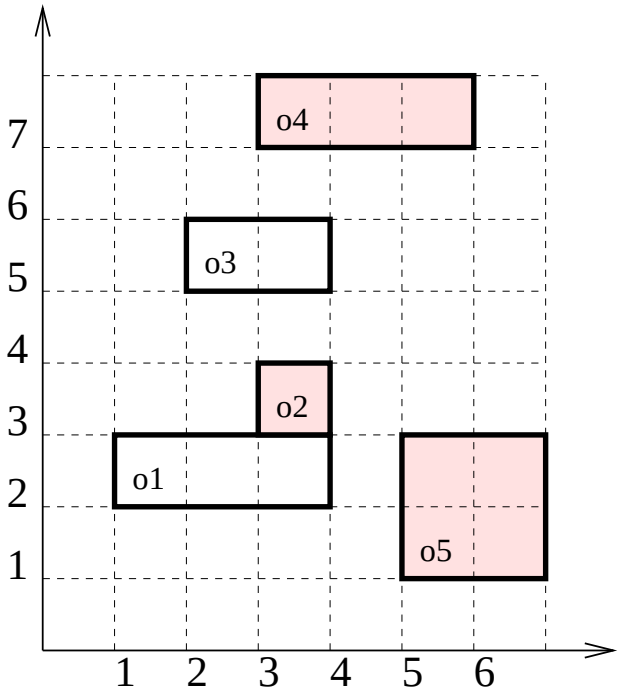

Figure 1: **Top.** The six macros and two rules of the running example. $ori(o,s,d)$ (resp. $end(o,s,d)$) stands for the origin (resp. end) in dimension d of object o with shape s. **Bottom.** One solution, where $s_3 = 32, x_{51} = 5, x_{52} = 1$. No objects overlap and objects of type 1 do not meet. Grey resp. white objects are of type 1 resp. of type 2.

1.1 Implementation overview

Fig. 2 provides the overall architecture of the implementation. When the *geost* constraint is posted, the given business rules are translated, first into a QFPA formula, then into generators of forbidden sets, *FS-generators* for short. Such generators are functions of the domain store. Finally, the FS-generators are translated into procedures that answer queries of the form "is there a forbidden orthotope that contains q" for a given $q \in \mathbb{Z}^k$. We call such procedures *FOQA procedures*.[2] Each time the constraint wakes up, the sweep-based algorithm [7] prunes the shape and coordinates of all relevant objects o. This is done my searching the Cartesian product of the domains of $o.x$ for a point $q \in \mathbb{Z}^k$ for which the FOQA procedure answers "no" (i.e., q does not belong to any forbidden set).

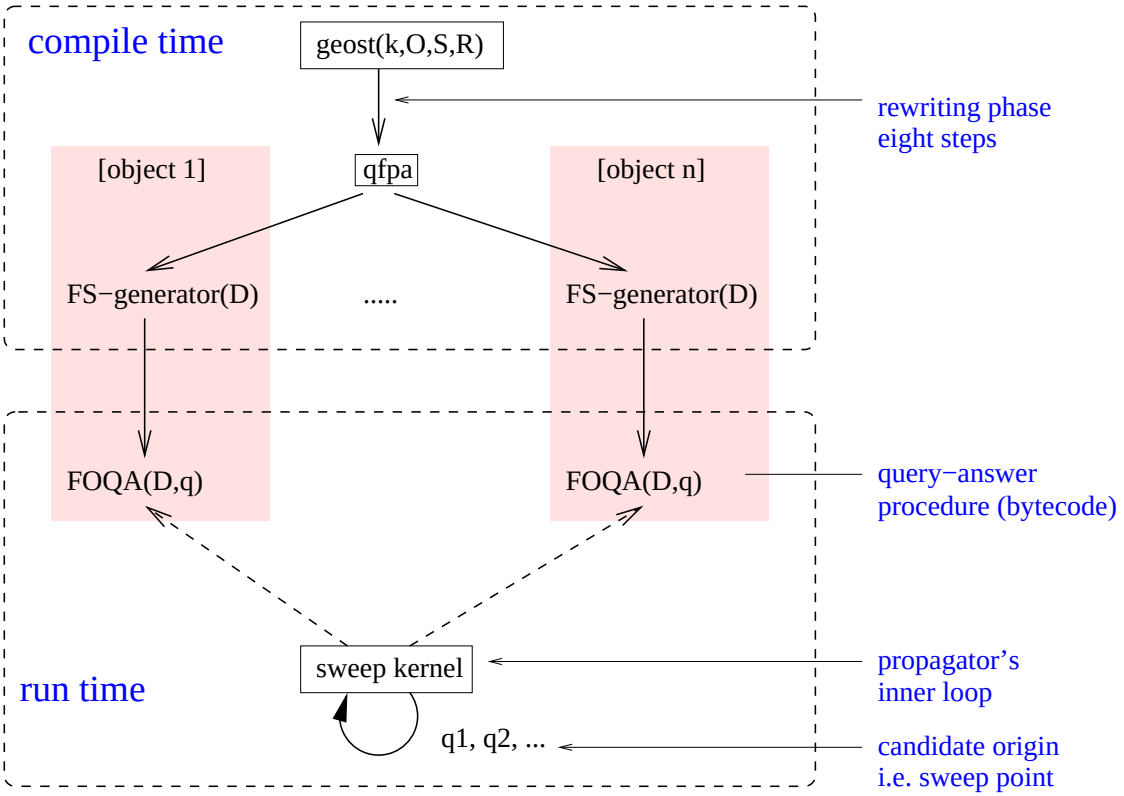

Figure 2: Overall architecture of the implementation. At the heart of the filtering algorithm is a sweep-point kernel, which prunes the coordinate of the objects, one object at a time. The kernel asks the object's FOQA procedure whether there is a forbidden orthotope that contains the current sweep point $q \in \mathbb{Z}^k$, until it gets the answer "no", or until the placement space has been exhausted. The FOQA procedure is a function of q as well as of the domain store.

1.2 Report outline

In Section 2, we present the rule language, its abstract syntax and its features. In Section 3, we present the QFPA core fragment of the language, its declarative semantics, and how the rule language is rewritten into QFPA. In Section 4, we describe (1) how a QFPA formula is compiled first to FS-generators and then to FOQA procedures, and (2) how a sweep-based filtering algorithm queries such procedures in order to prune the coordinates of all objects. In Section 5, we extend the filtering to accommodate polymorphic objects. In Section 6, we provide experimental evidence for search space reduction due to the global treatment of side constraints from five benchmarks: a placement problem from a car manufacturer, a packing-unpacking

[2] Forbidden Orthotope Query-Answer procedure.

problem, a square tiling problem, the vessel loading problem [9] and floor planning problems [10]. Before concluding, in Section 7, we mention a number of issues that we are currently working on.

2 The Rule Language

We now describe the syntax and features of the rule language. The syntax descriptions are kept abstract, with inductive definitions of legal terms instead of BNF grammars of legal sentences. The inductive definitions do use BNF-like notation. Fig. 3 shows the inductive definition of the rule language. A *macro* is simply a shorthand device: during a rewriting phase, whenever an expression matching the left-hand side of a macro is encountered, it is replaced by the corresponding right-hand side. A *fol* is a first-order logic formula that must hold for the constraint to be true. A *term* is a *variable*, an *integer*, an *identifier*, or a *compound term*. A *compound term* consists of a *functor* (an identifier) and one or more arguments (terms). A term is *ground* if it is free of variables. An *entity* denotes an object resp. a shifted box, which has a set of integer-valued attributes, where coordinates and shape ids can be domain variables. An *attref* is a reference to an attribute of an entity.

Bounded existential resp. *universal quantifiers* are provided. They are meaningful if the quantified variable occurs in the quantified *fol*. They are treated by expansion to a disjunction resp. a conjunction of instances of that *fol* where each element of the *collection* is substituted for the quantified variable. For example, formulas (1) and (2) below are equivalent:

$$\forall(x, [0, 1, 2], p(x)) \tag{1}$$

$$p(0) \wedge p(1) \wedge p(2) \tag{2}$$

In the context of $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$, quantified variables typically vary over a collection of dimensions, objects, or shifted boxes. $\texttt{objects}(\texttt{S})$ is a shorthand for the subset of $\mathcal{O}$ with object id in S. Similarly, $\texttt{sboxes}(\texttt{S})$ is a shorthand for the subset of $\mathcal{S}$ with shape id in S.

A *cardinality formula* specifies a variable quantified over a list of terms, a lower and an upper bound, and a *fol* template mentioning the quantified variable. The formula is true if and only if the number of true instances of the *fol* template is within the given bounds. Cardinality formulas [13] are treated by expansion to $\neg$, $\wedge$ and $\vee$ connectives [14]. For example, formulas (3) and (4) below are equivalent:

$$\#(y, [o_1, o_2, o_3], 2, 3, y.\text{type} > 5) \tag{3}$$

$$\bigvee \begin{pmatrix} o_1.\text{type} > 5 \wedge o_2.\text{type} > 5 \\ o_1.\text{type} > 5 \wedge o_3.\text{type} > 5 \\ o_2.\text{type} > 5 \wedge o_3.\text{type} > 5 \end{pmatrix} \tag{4}$$

Arithmetic expressions and *comparisons* are over the rational numbers.

The rationale for this is that business rules are often expressed in terms of fractions of measures like weight or volume. However, such fractions are converted to integers during rewriting.

A *folding operator* allows to express e.g. the sum of some attribute over a set of objects. The operator specifies a variable quantified over a list of terms, a binary operator, an identity element, and a template mentioning the quantified variable. The identity element is needed for the empty list case. For example, formulas (5) and (6) below are equivalent:

$$@(y, [o_1, o_2, o_3], +, 0, y.\text{weight}) \tag{5}$$

$$o_1.\text{weight} + o_2.\text{weight} + o_3.\text{weight} \tag{6}$$

sentence	::=	*macro* | *fol*	
macro	::=	$head \Longrightarrow body$	
head	::=	*term*	{ to be substituted by a *body* }
body	::=	*term*	{ to substitute for a *head* }
fol	::=	$\neg fol$	{ negation }
	|	$fol \wedge fol$	{ conjunction }
	|	$fol \vee fol$	{ disjunction }
	|	$fol \Rightarrow fol$	{ implication }
	|	$fol \Leftrightarrow fol$	{ equivalence }
	|	$\exists(var, collection, fol)$	{ existential quantification }
	|	$\forall(var, collection, fol)$	{ universal quantification }
	|	$\#(var, collection, integer, integer, fol)$	{ cardinality }
	|	`true`	
	|	`false`	
	|	*expr relop expr*	{ arith. comparison over $\mathbb{Q}$ }
	|	*head*	{ macro application }
expr	::=	$expr + expr$	
	|	$expr - expr$	
	|	$\min(expr, expr)$	
	|	$\max(expr, expr)$	
	|	$expr \times groundexpr$	
	|	$groundexpr \times expr$	
	|	$expr / groundexpr$	
	|	*attref*	
	|	*integer*	
	|	$@(var, collection, fop, expr, expr)$	{ folding }
	|	*variable*	{ quantified variable }
	|	*head*	{ macro application }
groundexpr	::=	*expr*	{ where *expr* is ground }
attref	::=	$entity.attr$	
attr	::=	*term*	{ attribute name }
	|	*variable*	{ quantified variable }
relop	::=	$< \mid = \mid > \mid \neq \mid \leq \mid \geq$	
fop	::=	$+ \mid \min \mid \max$	
collection	::=	*list*	
	|	`objects`$(list)$	{ list of oids }
	|	`sboxes`$(list)$	{ list of sids }
list	::=	$[] \mid [term \mid list]$	

Figure 3: The rule language.

3 QFPA Core Fragment

In this section, we show how a formula in the rule language is rewritten by a series of equivalence-preserving transformations into a *qfpa*, i.e. a formula of the core fragment of the language shown in Fig. 4. In fact, the fragment coincides with Quantifier-Free Presburger Arithmetic (QFPA), although QFPA is usually described with a less restrictive syntax. The declarative semantics of a *qfpa* is the natural one.

$qfpa$	$::=$	$qfpa \wedge qfpa$	{ conjunction }
	$\mid$	$qfpa \vee qfpa$	{ disjunction }
	$\mid$	$\sum_i integer_i \cdot attref_i \geq integer$	{ integer linear inequation }

Figure 4: Core fragment of the language. An *attref* corresponds to a nonground attribute of an object or an attribute of a shifted box of a polymorphic object.

QFPA is widely used in symbolic verification, and there has been much work on deciding whether a given QFPA formula is satisfiable [15]. Many methods based on integer programming techniques [16] rely on having the formula on disjunctive normal form. However, for constraint programming purposes, we are interested in necessary conditions that can be used for filtering domain variables, and we are not aware on any such work on QFPA. In [17], filtering algorithms for logical combinations of ad-hoc constraints[3] are proposed, but it is not clear whether that approach can be extended to QFPA. For that, we would need to provide supports of *qfpas*.

3.1 Rewriting into QFPA

We now show the details of rewriting the formula given as the *geost* parameter $\mathcal{R}$ in the following eight steps into a *qfpa* $\hat{\mathcal{R}}$. Fig. 5 shows the details of some of these steps as tables. The cell in the column entitled **condition**, if nonempty, mentions the condition under which the rewrite is done. We will later show how $\hat{\mathcal{R}}$ is translated to FS-generators.

1. **Macro expansion and constant folding.** The implication and equivalence connectives, bounded quantifiers, and cardinality and folding operators are eliminated. Ground integer expressions are replaced by their values. Collection shorthand are expanded. See Fig. 5, top.

2. **Elimination of negation.** Using DeMorgan's laws and negating relevant *relop*s.

3. **Normalization of arithmetic.** Arithmetic relations are normalized to one of the forms $expr \geq 0$ or $expr > 0$. See Fig. 5, bottom left.

4. **Elimination of $\times$, $/$ and $-$.** Any occurrence of these operators in arithmetic expressions is eliminated. At the same time, all operands are associated with a rational coefficient (c in Fig. 5, bottom center). The elimination is made possible by the fact that in multiplication, at least one factor must be ground and is simply multiplied into the coefficient. Similarly, in division, the coefficient is simply divided by the divisor, which must be ground. After this step, an arithmetic expression is:

   - a rational number c, denoted $c \cdot 1$, or
   - an attribute reference r with a rational coefficient c, denoted $c \cdot r$, or
   - two arithmetic expressions combined with $+$, $\min$ or $\max$.

5. **Moving $+$ inside $\min$ and $\max$.** Any expression with $\min$ or $\max$ occurring inside $+$ are rewritten by using the commutative and distributive laws (7) so that the $+$ is moved inside the other operator.

[3] Also known as constraints given in extension.

$$\begin{aligned} a + b &= b + a \\ a + \min(b, c) &= \min(a + b, a + c) \\ a + \max(b, c) &= \max(a + b, a + c) \end{aligned} \tag{7}$$

6. **Elimination of** $\min$ **and** $\max$**.** Any $\min$ or $\max$ operators occurring in arithmetic relations are eliminated, replacing such relations by new relations combined by $\wedge$ or $\vee$. After this step, an arithmetic expression is a linear combination of *attref*s with rational coefficients, plus an optional constant. See Fig. 5, bottom right.

7. **Elimination of rational numbers.** Any arithmetic relation r, which can now only be of the form $e > 0$ or $e \geq 0$, is normalized into the form $e'' \geq c''$ where e'' and c'' are obtained as follows:

   - Let e' be the linear combination obtained by multiplying e by the least common multiplier of the denominators of the coefficients of e. Recall that those coefficients are rational numbers. Thus, the coefficients of e' are integers.
   - Let c' be 1 if r is of the form $e > 0$, or 0 if r is of the form $e \geq 0$.
   - If e' contains a constant term c, then $e'' = e' - c$ and $c'' = c' - c$. Otherwise, $e'' = e'$ and $c'' = c'$.

8. **Simplification.** Any entailed or disentailed arithmetic comparison is replaced by the appropriate logical constant (**true** or **false**). Any $\wedge$ or $\vee$ expression containing one of these constants is simplified using partial evaluation.

Example 2 *Returning to our running example, we show in Figs. 6-7 how the initial business rules of 1 are successively rewritten into a* qfpa*. The example shows that the rewrite process essentially amounts to partial evaluation. The resulting* qfpa $\hat{\mathcal{R}}$ *is a conjunction of six subformulas corresponding respectively to:*

- *From the business rule* `all_not_overlap_sboxes`*, conditions to prevent* o_5 *from overlapping* o_1*,* o_2*,* o_3 *and* o_4*.*
- *From the business rule* `all_type1_not_meet_sboxes`*, conditions to prevent* o_5 *from meeting* o_2 *and* o_4*.*

Note that the rules among the fixed objects o_1*,* o_2 *and* o_4 *are rewritten away by partial evaluation, and do not appear in the* qfpa*.*

□

4 Compiling to an Efficient Run-Time Representation

It is straightforward to obtain necessary conditions for *qfpa*s as well as pruning rules operating on one variable at a time. Based on such conditions and pruning rules, we will show how to construct first FS-generators and then FOQA procedures. Such generators can be seen a generalization of the indexicals of cc(FD) [18]. Finally, we show how a sweep-based filtering algorithm queries such procedures in order to prune the coordinates of all objects.

Indexicals have been used in the context of CLP(FD) [19, 20], AKL [21], finite set constraints [22] and ad-hoc constraints [23]. They have proven a powerful and efficient way of implementing constraint propagation. A key feature of an indexical is that it is a function of the current domains of the variables on which it depends. Thus, indexicals also capture the propagation from variables to variables that occurs as variables are pruned. In the cited implementations, an indexical is a procedure that computes the feasible set of values for a variable. We generalize this notion to generating a forbidden set of k-dimensional points for the origin of an object. Our FS-generators capture the propagation from objects to objects that occurs as coordinates attributes are pruned.

line	$\mathbf{p}$	$\mathbf{R_1(p)}$	condition
1	p	$R_1(q)$	$q = \text{macro}(p)$
2	$\neg p$	$\neg R_1(p)$	
3	$p \Rightarrow q$	$R_1(q \vee \neg p)$	
4	$p \Leftrightarrow q$	$R_1((p \Rightarrow q) \wedge (q \Rightarrow p))$	
5	$\exists(x, [y_1, \ldots, y_n], p)$	$R_1(p_{x/y_1} \vee \cdots \vee p_{x/y_n})$	
6	$\forall(x, [y_1, \ldots, y_n], p)$	$R_1(p_{x/y_1} \wedge \cdots \wedge p_{x/y_n})$	
7	$@(x, [y_1, \ldots, y_n], \circ, z, p)$	$R_1(p_{x/y_1} \circ \cdots \circ p_{x/y_n} \circ z)$	
8	$\#(x, [], l, u, p)$	`true`	$l \leq 0 \leq u$
9	$\#(x, [], l, u, p)$	`false`	$l > 0 \vee 0 > u$
10	$\#(x, [y_1, \ldots, y_n], l, u, p)$	$R_1\left(\begin{array}{l}(p_{x/y_1} \wedge \#(x, [y_2, \ldots, y_n], l-1, u-1, p) \vee \\ (\neg p_{x/y_1} \wedge \#(x, [y_2, \ldots, y_n], l, u, p)\end{array}\right)$	$n > 0$
11	*expr*	i	$i = \text{ieval}(p)$
12	`objects`$([\mathtt{o_1}, \ldots, \mathtt{o_n}])$	objects with the given oids	
13	`sboxes`$([\mathtt{s_1}, \ldots, \mathtt{s_n}])$	sboxes with the given sids	

$\mathbf{p}$	$\mathbf{R_3(p)}$
$x < y$	$y - x > 0$
$x > y$	$x - y > 0$
$x \leq y$	$y - x \geq 0$
$x \geq y$	$x - y \geq 0$
$x = y$	$x - y \geq 0 \wedge y - x \geq 0$
$x \neq y$	$x - y > 0 \vee y - x > 0$

$\mathbf{p}$	$\mathbf{R_6(p)}$
$\max(x, y) > 0$	$x > 0 \vee y > 0$
$\min(x, y) > 0$	$x > 0 \wedge y > 0$
$\max(x, y) \geq 0$	$x \geq 0 \vee y \geq 0$
$\min(x, y) \geq 0$	$x \geq 0 \wedge y \geq 0$

$\mathbf{p}$	$\mathbf{R_4(p, c)}$	condition
$\min(x, y)$	$\min(R_4(x, c), R_4(y, c))$	$c > 0$
$\min(x, y)$	$\max(R_4(x, c), R_4(y, c))$	$c < 0$
$\max(x, y)$	$\max(R_4(x, c), R_4(y, c))$	$c > 0$
$\max(x, y)$	$\min(R_4(x, c), R_4(y, c))$	$c < 0$
$x + y$	$R_4(x, c) + R_4(y, c)$	
$x - y$	$R_4(x, c) + R_4(y, -c)$	
$x \times y$	$R_4(x, c \times v)$	$v = \text{reval}(y)$
$x \times y$	$R_4(y, c \times v)$	$v = \text{reval}(x)$
x/y	$R_4(x, c/v)$	$v = \text{reval}(y)$
x	$(c \times x) \cdot 1$	x integer
x	$c \cdot x$	x *attref*

Figure 5: **Top.** Rewrite phase 1, of a formula p into a formula $R_1(p)$, eliminates macros (line 1), implication (line 3), equivalence (line 4), bounded quantifiers (line 5-6), folding operators (line 7), cardinality operators (line 8-10), ground attribute references (line 11), and entity collections (line 12-13). If a compound term does not match any line 1-13, its arguments are rewritten recursively. $p_{x/y}$ denotes the term p with y substituted for x. $\text{macro}(p)$ denotes the macro expansion of the formula p. $\text{ieval}(p)$ denotes the integer value of the ground expression p. **Bottom left top.** Rewrite phase 3, of a formula p into a formula $R_3(p)$, normalizes comparison operators into either $\geq$ or $>$. **Bottom right.** Rewrite phase 4, of a formula p into a formula $R_4(p, 1)$, eliminates the $-$, $\times$ and $/$ operators, and assigns a coefficient c to each operand of the rewritten formula. $\text{reval}(y)$ denotes the rational value of the ground expression y. **Bottom left bottom.** Rewrite phase 6, of a formula p into a formula $R_6(p)$, eliminates $\min$ and $\max$.

```
all_not_overlap_sboxes([1,2],[1,2,3,4,5]),
all_type1_not_meet_sboxes([1,2],[1,2,3,4,5])]).
```

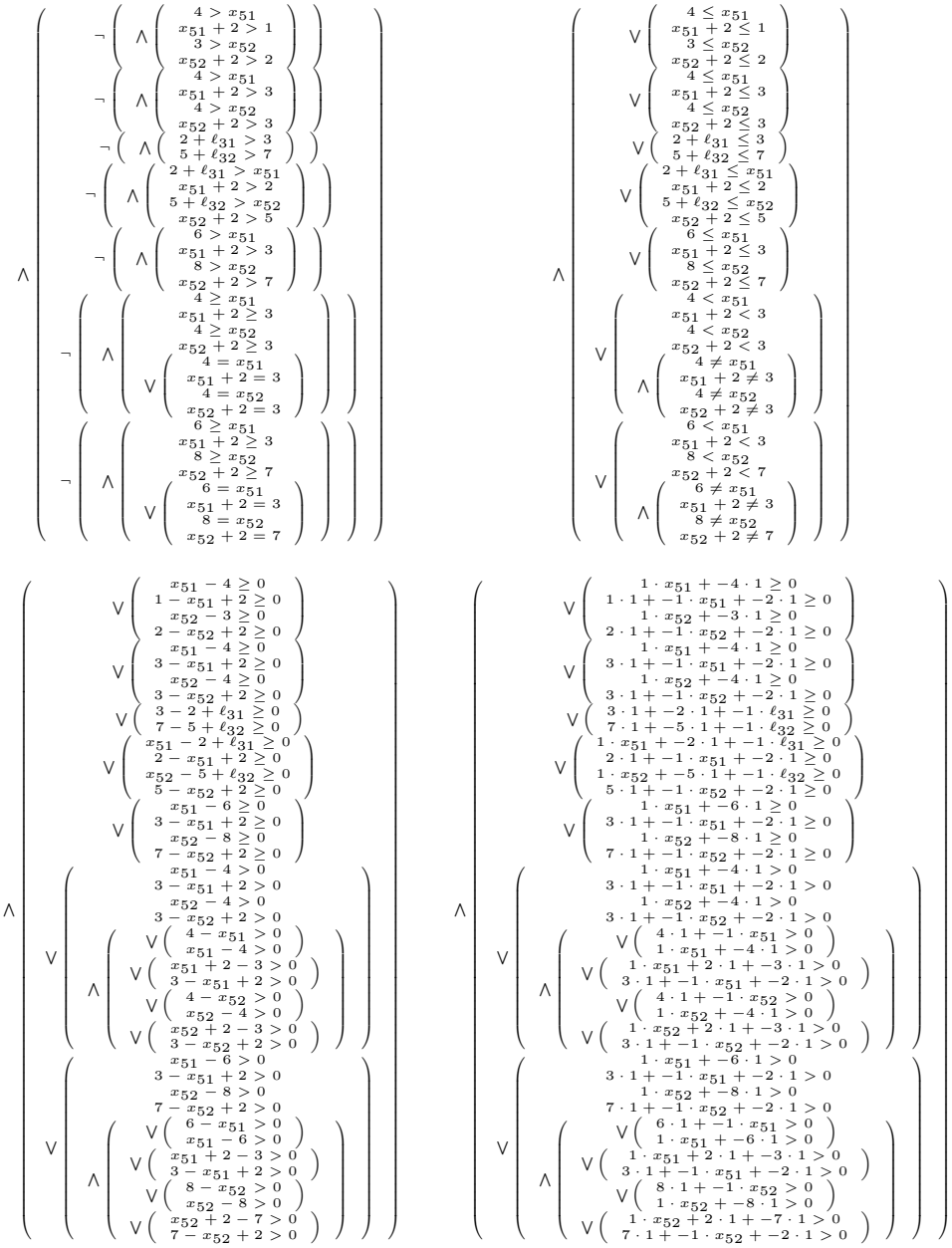

Figure 6: Running example business rules (**top**), formula after macro expansion and constant folding (step 1, **middle left**), elimination of negation (step 2, **middle right**), normalization of arithmetic (step 3, **bottom left**), and elimination of operators (step 4, **bottom right**).

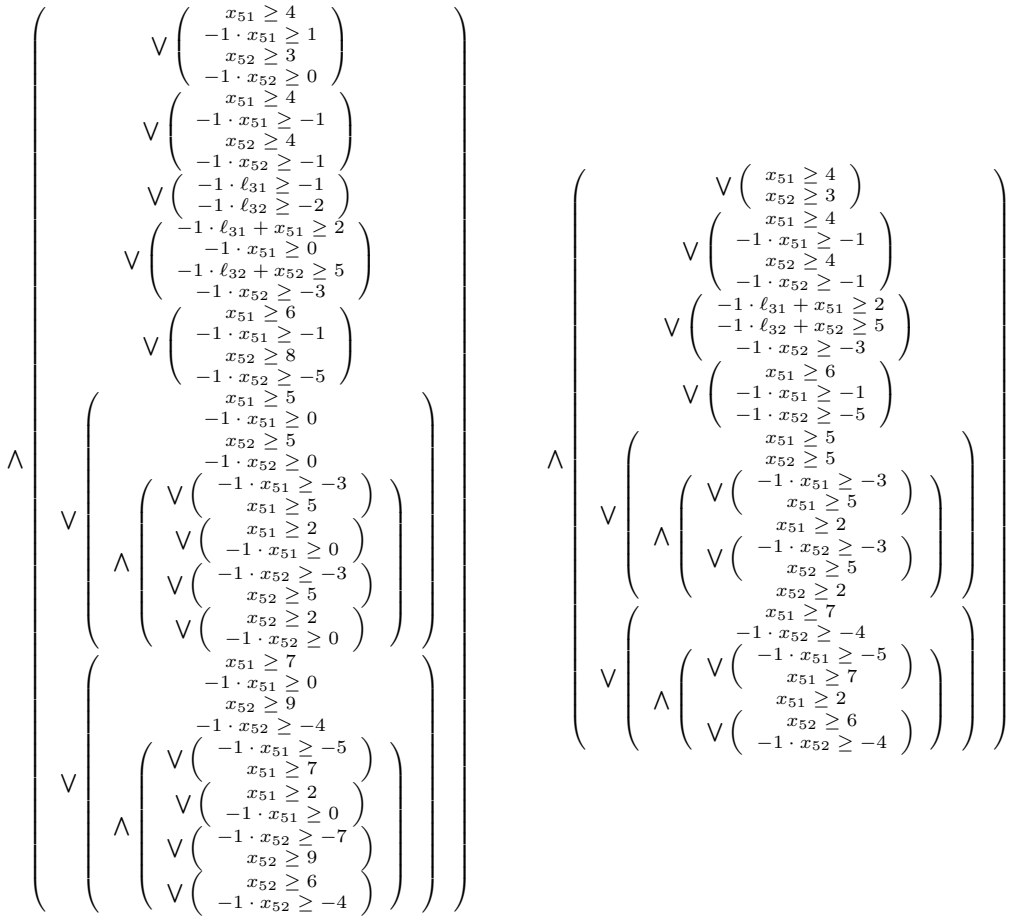

Figure 7: Running example formula after elimination of rational numbers (step 7, **left**) and simplification (step 8, **right**), resulting in a QFPA formula $\hat{\mathcal{R}}$. Steps 5-6 were not needed, since our example did not use $\min$ or $\max$.

4.1 Necessary Conditions

For a formula R denoting a linear combination of variables, let $MAX(R)$ denote the expression that replaces every *attref* x in R by $\overline{x}$ if x occurs with a positive coefficient, and by $\underline{x}$ otherwise. Thus, $MAX(R)$ is a formula that computes an upper bound of R wrt. the current domains.

We will ignore the degenerate cases where $\hat{\mathcal{R}}$ is `true` resp. `false`, in which case *geost* merely succeeds resp. fails. For the normal *qfpa* cases, we obtain the necessary conditions shown in Table 1.

qfpa t	**necessary condition** $N(t)$
$\sum_i c_i \cdot x_i \geq u$	$MAX(\sum_i c_i \cdot x_i) \geq u$
$p \vee q$	$N(p) \vee N(q)$
$p \wedge q$	$N(p) \wedge N(q)$

Table 1: Necessary condition $N(t)$ for *qfpa* t.

4.2 Pruning Rules

For the base case $\sum_i c_i \cdot x_i \geq u$, we have the well-known pruning rules (8), which provide sharp bounds; see e.g. [24] for details.

$$\forall j \begin{cases} x_j \geq \lceil \frac{u - MAX(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil, & \text{if } c_j > 0 \\ x_j \leq \lfloor \frac{-u + MAX(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor, & \text{otherwise} \end{cases} \tag{8}$$

Consider now a disjunction $p \vee q$ of two base cases and a variable x_j occurring in at least one disjunct.

- If x_j occurs in p but not in q, rule (8) is only valid for p if the necessary condition for q does not hold.
- Similarly if x_j occurs in q but not in p.
- If x_j occurs in both p and q, we can use rule (8) for both p and q and conclude that x_j must be in the union of the two feasible intervals.

Finally, consider a conjunction $p \wedge q$, i.e. both p and q must hold. If x_j occurs in both p and q, we can use rule (8) for both p and q and conclude that x_j must be in the intersection of the two feasible intervals.

Example 3 *Returning to our running example, consider the fragment* $x_{51} \geq 4 \vee x_{52} \geq 3$ *of the* qfpa*, which comes from a rule preventing* o_5 *from overlapping* o_1*. Suppose that we want to prune* x_{52}*. Then we can combine the necessary condition for* $x_{51} \geq 4$ *with rule (8) for* $x_{52} \geq 3$ *into the conditional pruning rule:*

$$\overline{x_{51}} < 4 \Rightarrow x_{52} \geq 3$$

However, as we will show in the next section, instead of using such conditional pruning rules, we unify necessary conditions and pruning rules into multi-dimensional forbidden sets and aggregate them per object. For the above fragment, the two-dimensional forbidden set for o_5 *is* $([1,1],[3,2])$*, which means that the following should hold:*

$$(x_{51}, x_{52}) \notin \{(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)\}$$

□

4.3 FS-Generators and FOQA procedures

Recall that the set of rules given in $\mathcal{R}$ has been rewritten into a *qfpa* $\hat{\mathcal{R}}$. The idea is to compile this *qfpa*, for each object o mentioned by it, into an FS-generator for $\hat{\mathcal{R}}$ and o, which can generate forbidden sets to be aggregated and used by the sweep-point kernel [7] to prune the nonground attributes of o. However, this idea has the flaw that the forbidden sets are generally computed as k-dimensional half-planes combined by nested union and intersection operations. The cost of this computation can blow up exponentially in space as well as in time. Recall that the sweep-point kernel maintains a current sweep point in $q \in \mathbb{Z}^k$ and repeatedly queries "is there a forbidden orthotope that contains q", updating q until it gets the answer "no". But the sweep-point kernel does not need to know all forbidden points in advance. It repeatedly checks whether or not the current sweep point belongs to some forbidden orthotope, and if so, compute from that orthotope how to update the sweep point in future moves. This is precisely what an FOQA procedures provides. Moreover, an FOQA procedure runs in space and time linear in the size of the *qfpa*. Now let us explore the details of these ideas.

Definition 1 *An* FS-generator for a qfpa r and an object o *is a procedure that functions as a generator of forbidden sets for r and o. It is of the form $o.x \notin \mathrm{FSG}(r, o)$ where $\mathrm{FSG}(r, o)$ is defined in Fig. 8. The object o* depends on object o' *if $ibody$ mentions o'.*

FS-generators are described by the inductive definition shown in Fig. 8. They are built up from generators of k-dimensional half-planes, combined by union and intersection operations. Throughout, we will use expressions $(f.\min, f.\max)$ denoting an orthotope with lower bound vector $f.\min$ and an upper bound vector $f.\max$. This is a shorthand for the intersection of $2 \cdot k$ k-dimensional half-planes.

$\mathit{FS\text{-}generator}$	::=	$\mathit{object}.x \notin \mathit{ibody}$	
ibody	::=	$\mathit{ibody} \cap \mathit{ibody}$	
	|	$\mathit{ibody} \cup \mathit{ibody}$	
	|	$\{p \in \mathbb{Z}^k \mid p[d] < \lceil \frac{\mathit{integer} - \sum \mathit{ubterm}}{\mathit{usi}} \rceil\}$	
	|	$\{p \in \mathbb{Z}^k \mid p[d] > \lfloor \frac{\mathit{integer} + \sum \mathit{ubterm}}{\mathit{usi}} \rfloor\}$	
	|	$\mathbf{if}\ \sum \mathit{ubterm} < \mathit{integer}\ \mathbf{then}\ \mathbb{Z}^k\ \mathbf{else}\ \emptyset$	
ubterm	::=	$\mathit{usi} \cdot \overline{\mathit{attref}}$	
	|	$-\mathit{usi} \cdot \underline{\mathit{attref}}$	
	|	$\mathit{integer}$	
d	::=	usi	{ denoting a dimension }
usi	::=	$\mathit{integer}$	$\{ > 0 \}$

Figure 8: FS-Generators. Any object o' occurring in $ibody$ generates a dependency of $object$ on o'.

Definition 2 *An* FOQA procedure for a qfpa r, an object o, and a point $q \in \mathbb{Z}^k$ *is a procedure that returns:*

- *f, where $f \neq \emptyset$ is a forbidden orthotope for r and o and $q \in f$, or*
- *$\emptyset$, if no such forbidden orthotope exists.*

4.4 Compilation

For each object $o \in \mathcal{O}$, the *qfpa* $\hat{\mathcal{R}}$ is mapped to a FS-generator of the form $o.x \notin \mathrm{FSG}(\hat{\mathcal{R}}, o)$. The mapping is done according to Table 2 and closely follows the pruning rules (8), except now we want to obtain a

forbidden set instead of a feasible interval. Rows 1-2 of Table 2 are analogous to the recursive computation of inconsistent assignments in [11, Table 1]. The expressions in column 2 of rows 3-4 correspond to half-planes. We will use a shorthand notation $(f.\min, f.\max)$ for intersections of such half-planes. Row 5 corresponds to the case where an integer linear inequation r does not mention o, in which case all points are forbidden for o if r is disentailed, and no points are forbidden for o otherwise.

r	$\mathrm{FSG}(\mathbf{r}, \mathbf{o})$	**condition**
$r_1 \wedge r_2$	$\mathrm{FSG}(r_1, o) \cup \mathrm{FSG}(r_2, o)$	
$r_1 \vee r_2$	$\mathrm{FSG}(r_1, o) \cap \mathrm{FSG}(r_2, o)$	
$\sum_i c_i \cdot x_i \geq u$	$\{p \in \mathbb{Z}^k \mid p[d] < \lceil \frac{u - MAX(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil\}$	$x_j = o.x[d], c_j > 0$
$\sum_i c_i \cdot x_i \geq u$	$\{p \in \mathbb{Z}^k \mid p[d] > \lfloor \frac{-u + MAX(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor\}$	$x_j = o.x[d], c_j < 0$
$\sum_i c_i \cdot x_i \geq u$	$\mathbf{if}\ MAX(\sum_i c_i \cdot x_i) < u\ \mathbf{then}\ \mathbb{Z}^k\ \mathbf{else}\ \emptyset$	$o.x[d] \notin \{x_i\}$

Table 2: Mapping a *qfpa* r to an FS-generator, $\mathrm{FSG}(r, o)$, for the object o. We assume here that o is not polymorphic.

It is worth noting that our FS-generators do not necessarily compute sets that include all infeasible points. Consider e.g. the formula $o.x[1] + o.x[2] = 3$ and its corresponding *qfpa*:

$$o.x[1] + o.x[2] \geq 3 \wedge -o.x[1] - o.x[2] \geq -3$$

which we compile to the FS-generator:

$$o.x \notin \left\{ (x, y) \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} x < 3 - \overline{o.x[2]} \vee \\ x > 3 - \underline{o.x[2]} \vee \\ y < 3 - \overline{\overline{o.x[1]}} \vee \\ y > 3 - \underline{o.x[1]} \end{array} \right\} \tag{9}$$

For example, with the domains $o.x[1] \in [0, 2], o.x[2] \in [0, 2]$, (9) would generate the forbidden set:

$$\{(x, y) \in \mathbb{Z}^2 \mid x < 1 \vee x > 2 \vee y < 1 \vee y > 2\}$$

which does not include the two forbidden points $(1, 1)$ and $(2, 2)$. But this does not make the filtering unsound, for when the sweep-point kernel, while pruning o, queries the FOQA procedure obtained from (9), it will do so under the assumption that $o.x$ is assigned to the current sweep-point, in which case the FOQA procedure will return $\emptyset$ if and only if $o.x[1] + o.x[2] = 3$ holds.

As shown in Table 3, the FS-generators are mapped to FOQA procedures. When a *geost* constraint is posted, the relevant FOQA procedures are computed and encoded as bytecode for a register-based virtual machine that is invoked for each query by the sweep-point kernel. The procedures tend to contain a lot of common subexpressions, which can be conveniently eliminated in a register-based virtual machine. Dependency information is also computed for use by $\mathrm{FilterCtrs}$ (see Section 4.5) and stored with each FOQA procedure.

Example 4 *Returning to our running example, we obtained a* qfpa *which was a conjunction of six subformulas (see Fig. 7). Ignoring the three ground objects, we compute FS-generators for object o_3 (10) and object o_5 (11). Finally, we obtain from (11) the FOQA procedure shown in Algorithm 1. No FOQA procedure is generated from (10), for our current sweep-point kernel is always invoked with a specific shape id for the object being pruned. For a polymorphic object o, first the kernel is invoked with the assumption $o.sid = s$ for each shape $s \in o.sid$, and finally attribute values that were pruned away for every shape are pruned away from o, possibly including specific shape ids.*

The ability to prune shape ids inside the sweep-point kernel is being considered as a future development. The FS-generators reflect the following business rules:

$\mathbf{g}$	$[\mathbf{g}]_\mathbf{q}$
$g_1 \cup g_2$	**if** $[g_1]_q \neq \emptyset$ **then** $[g_1]_q$ **else** $[g_2]_q$
$g_1 \cap g_2$	$[g_1]_q \cap [g_2]_q$
$\{p \in \mathbb{Z}^k \mid p[d] < ub\}$	**if** $q[d] < ub$ **then** g **else** $\emptyset$
$\{p \in \mathbb{Z}^k \mid p[d] > lb\}$	**if** $q[d] > lb$ **then** g **else** $\emptyset$
if $u < v$ **then** $\mathbb{Z}^k$ **else** $\emptyset$	g

Table 3: Mapping an FS-generator, g, for *qfpa* r and object o to $[g]_q$, a FOQA procedure for r, o and $q \in \mathbb{Z}^k$.

o_3 *must not take a shape that will cause it to overlap* o_5*. This FS-generator encodes the fact that* o_3 *depends on* o_5*. Initially, no forbidden boxes are generated.*

o_5 *must not overlap any object, nor meet* o_2 *nor* o_4*. This FS-generator encodes the fact that* o_5 *depends on* o_3*. It will initially generate the forbidden boxes shown in Fig. 9 (top).*

□

$$s_3 \notin \bigcap \begin{pmatrix} \{i \in \mathrm{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{31} > \overline{x_{51}} - 2\} \\ \{i \in \mathrm{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{32} > \overline{x_{52}} - 5\} \\ \textbf{if } \underline{x_{52}} > 3 \textbf{ then } \mathbb{Z} \textbf{ else } \emptyset \end{pmatrix} \tag{10}$$

$$o_5.x \notin \bigcup \begin{pmatrix} ([1,4],[\underline{\ell_{31}}+1, \underline{\ell_{32}}+4]) \\ ([1,1],[3,2]) \\ ([2,2],[3,3]) \\ ([2,6],[5,6]) \\ \bigcap \left(\begin{array}{c} ([1,1],[4,4]) \\ \bigcup \left(\begin{array}{c} ([4,1],[4,6]) \\ ([1,1],[1,6]) \\ ([1,4],[9,4]) \\ ([1,1],[9,1]) \end{array} \right) \end{array} \right) \\ \bigcap \left(\begin{array}{c} ([1,5],[6,6]) \\ \bigcup \left(\begin{array}{c} ([1,5],[9,5]) \\ ([6,1],[6,6]) \\ ([1,1],[1,6]) \end{array} \right) \end{array} \right) \end{pmatrix} \tag{11}$$

4.5 Filtering Algorithm

The use of sweep algorithms in constraint filtering algorithms was introduced in [25] and applied to the non-overlapping 2D rectangles constraints. In [7, Algorithm 1], we gave a filtering algorithm consisting of two procedures $\mathrm{FilterCtrs}$ and $\mathrm{PruneMin}$ for adjusting the lower and upper bounds of the coordinates of all object. In Algorithm 2, we present a filtering algorithm that is modified for handling rules: (a) we use FOQA procedures instead of precomputing sets of forbidden orthotopes, and (b) we use their dependency information in the fixpoint loop.

The main procedure, $\mathrm{FilterCtrs}$, is a fixpoint loop around $\mathrm{PruneMin}(o, d, k)$, which searches for the first point c, by lexicographic order wrt. dimensions $d, (d+1) \mod k, \ldots, (d-1) \mod k$, that is inside the domain of $o.x$ but not inside any forbidden region. If such a c exists, the algorithm sets $\underline{o.x[d]}$ to $c[d]$, otherwise it fails. Two state vectors are maintained: the *sweep point* c, which holds a candidate value for $o.x$, and the *jump vector* n, which records knowledge about encountered forbidden regions.

The algorithm starts its recursive traversal of the placement space at point $c = \underline{o.x}$ with $n = \overline{o.x} + 1$ and could in principle explore all points of the domains of $o.x$, one by one, in increasing lexicographic order

```
PROCEDURE FOQA(q)        // obtained from (11)
 1: r_1 ← ([1,4],[ℓ_31 + 1, ℓ_32 + 4])        // dependency on o_3
 2: if q ∈ r_1 then return r_1 endif        // (A)
 3: r_1 ← ([1,1],[3,2])
 4: if q ∈ r_1 then return r_1 endif        // (B)
 5: r_1 ← ([2,2],[3,3])
 6: if q ∈ r_1 then return r_1 endif        // (C)
 7: r_1 ← ([2,6],[5,6])
 8: if q ∈ r_1 then return r_1 endif        // (D)
 9: r_1 ← ([1,1],[4,4])
10: if q ∈ r_1 then
11:     r_2 ← ([4,1],[4,6])
12:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (E)
13:     r_2 ← ([1,1],[1,6])
14:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (F)
15:     r_2 ← ([1,4],[9,4])
16:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (G)
17:     r_2 ← ([1,1],[9,1])
18:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (H)
19: end if
20: r_1 ← ([1,5],[6,6])
21: if q ∈ r_1 then
22:     r_2 ← ([1,5],[9,5])
23:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (I)
24:     r_2 ← ([6,1],[6,6])
25:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (J)
26:     r_2 ← ([1,1],[1,6])
27:     if q ∈ r_2 then return r_1 ∩ r_2 endif        // (K)
28: end if
29: return ∅
```

Algorithm 1: FOQA procedure obtained from (11) for the running example. o_5 depends on o_3. Lines returning forbidden orthotope are annotated correspondingly with the forbidden orthotopes of the top part of Fig. 9.

wrt. dimensions $d, (d+1) \mod k, \ldots, (d-1) \mod k$, until the first desired point is found. To make the search efficient, it skips points that are known to be inside some forbidden region. This knowledge is encoded in n, which is updated for every new f (see line 5) recording the fact that new candidate points can be found beyond that value. Whenever we skip to the next candidate point, we reset the elements of n that were used to their original values (see lines 6–15).

Example 5 *Returning to our running example, suppose now that the sweep-point kernel wants to adjust the lower bound of x_{51}. Fig. 9 (bottom) traces the steps performed by the algorithm when it walks from a lexicographically smallest position to the first feasible position of o_5. The result is that the lower bound of x_{51} is adjusted to 5.*

□

5 Polymorphism

We say that an object o is *polymorphic* if its shape id is nonground. This feature could for example be used to model a crate that can be rotated 90 degrees around some axis, in which case each rotated position would correspond to a distinct shape.

In the context of configuration problems, polymorphism can also be used to model the fact that we have to select for an abstract object a possible concrete object that realizes a given function, e.g. selecting a table among different possible table models.

Polymorphism is not a semantic issue, as the declarative semantics is defined in terms of ground objects. But it is an issue for the operational semantics, i.e. for filtering. We now describe how a small extension to the implementation allows to deal with polymorphic objects.

Polymorphic shifted boxes. With polymorphic objects, the expanded sentences of the rule language will mention attributes of shifted boxes, where the values of those attributes depend on the shape id. To deal with this complication, we introduce for polymorphic objects o a virtual $pbox[j]$ attribute, which stands for the j^{th} shifted box that has the same shape id as o. Thus a *pbox* attribute behaves like a shifted box but with nonground attributes that have evaluable lower and upper bounds, which is precisely what is needed in order to use the necessary conditions (Table 1) and pruning rules (8). Phase 1 of the rewrite process introduces *pbox*es when it encounters an expression `sboxes`$([\texttt{o}.sid])$ and o is polymorphic. Assuming that each possible shape of o consists of the same number, n, of shifted boxes, the expression is rewritten to $[o.pbox[1], \ldots, o.pbox[n]]$. Thus the requirement that n be fixed is a restriction of the approach.

6 Experimental Results

The business rules were integrated into our *geost* kernel [7]. When a *geost* constraint is posted, the rules are transformed to FOQA procedures, which are transferred to the sweep-point kernel in bytecode representation for use during filtering. The transformation is done in Prolog whereas the kernel is written in C. The basic non-overlapping constraint as well as symmetry breaking lexicographic ordering constraints were handled by built-in methods of the kernel [8, 5]. All experiments were run in SICStus Prolog 4 compiled with gcc version 4.1.0 with the -O2 option on a 3GHz Pentium IV with 1MB of cache. Full benchmark details are given in the Appendix.

6.1 Benchmarks

Placement. We studied the placement problem given in Appendix A, which involves 9 objects to be placed inside a container under certain placement rules. This problem instance was modeled as a *geost* constraint over 9 objects, whose origin variables were fixed by greedy assignment [5] in 1100 msec. The sweep-point kernel issued 372 FOQA queries. The execution time was totally dominated by rule compilation; the time spent in greedy assignment disappeared in the noise. If the placement rules are expressed as constraints in conjunction with *geost*, greedy assignment has to be replaced by standard labeling search, which needed 672 backtracks. The execution time was 1150 msec.

PROCEDURE $\mathrm{FilterCtrs}(k, \mathcal{O}) : \mathrm{bool}$

1: $Q \leftarrow \mathcal{O}$
2: **while** $Q \neq \emptyset$ **do**
3: $o \leftarrow$ some element from Q
4: $Q \leftarrow Q \setminus \{o\}$
5: **for** $d \leftarrow 0$ **to** $k-1$ **do**
6: **if** $\neg\mathrm{PruneMin}(o, d, k) \vee \neg\mathrm{PruneMax}(o, d, k)$ **then**
7: **return false** // no feasible origin
8: **end if**
9: **end for**
10: **if** a coordinate of o was pruned **then**
11: $Q \leftarrow Q \cup \{o' \mid o'$ depends on $o\}$
12: **end if**
13: **end while**
14: **return true** // fixpoint reached

PROCEDURE $\mathrm{PruneMin}(o, d, k) : \mathrm{bool}$

1: $c \leftarrow \underline{o.x}$ // initial position of the point
2: $n \leftarrow \overline{o.x} + 1$ // upper limits+1 in the different dimensions
3: $f \leftarrow \mathrm{FOQA}(o, c)$ // check if c is infeasible
4: **while** $f \neq \emptyset$ **do**
5: $n \leftarrow \min(n, f.\max + 1)$ // maintain n as min of u.b. of forbidden regions
6: **for** $j \leftarrow k-1$ **downto** 0 **do**
7: $j' \leftarrow (j+d) \mod k$ // least significant dimension first
8: $c[j'] \leftarrow n[j']$ // use $n[j']$ to jump
9: $n[j'] \leftarrow \overline{o.x}[j'] + 1$ // reset $n[j']$ to max
10: **if** $c[j'] \leq \overline{o.x}[j']$ **then**
11: **goto** next // candidate point found
12: **else**
13: $c[j'] \leftarrow \underline{o.x}[j']$ // exhausted a dimension, reset $c[j']$
14: **end if**
15: **end for**
16: **return false** // no next candidate point
17: next: $f \leftarrow \mathrm{FOQA}(o, c)$ // check again if c is infeasible
18: **end while**
19: $\underline{o.x}[d] \leftarrow c[d]$ // adjust lb of origin in dim. d
20: **return true**

Algorithm 2: $\mathrm{FilterCtrs}$ is the filtering algorithm of $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$. It maintains a set Q of objects that need filtering. Whenever an object o is pruned, all objects that depend on o are added to this set. When the set becomes empty, the fixpoint is reached. $\mathrm{PruneMin}$ adjusts the lower bound $o.x[d]$. Since $\mathrm{PruneMax}$ is similar, it is omitted. $\mathrm{FOQA}(o, c)$ returns a forbidden orthotope $f \neq \emptyset$ for the *qfpa* derived from the rules given to *geost* and o, where $c \in f$, is such an f exists, and $\emptyset$ otherwise. f is represented represented as a lower bound vector $f.\min$ and an upper bound vector $f.\max$.

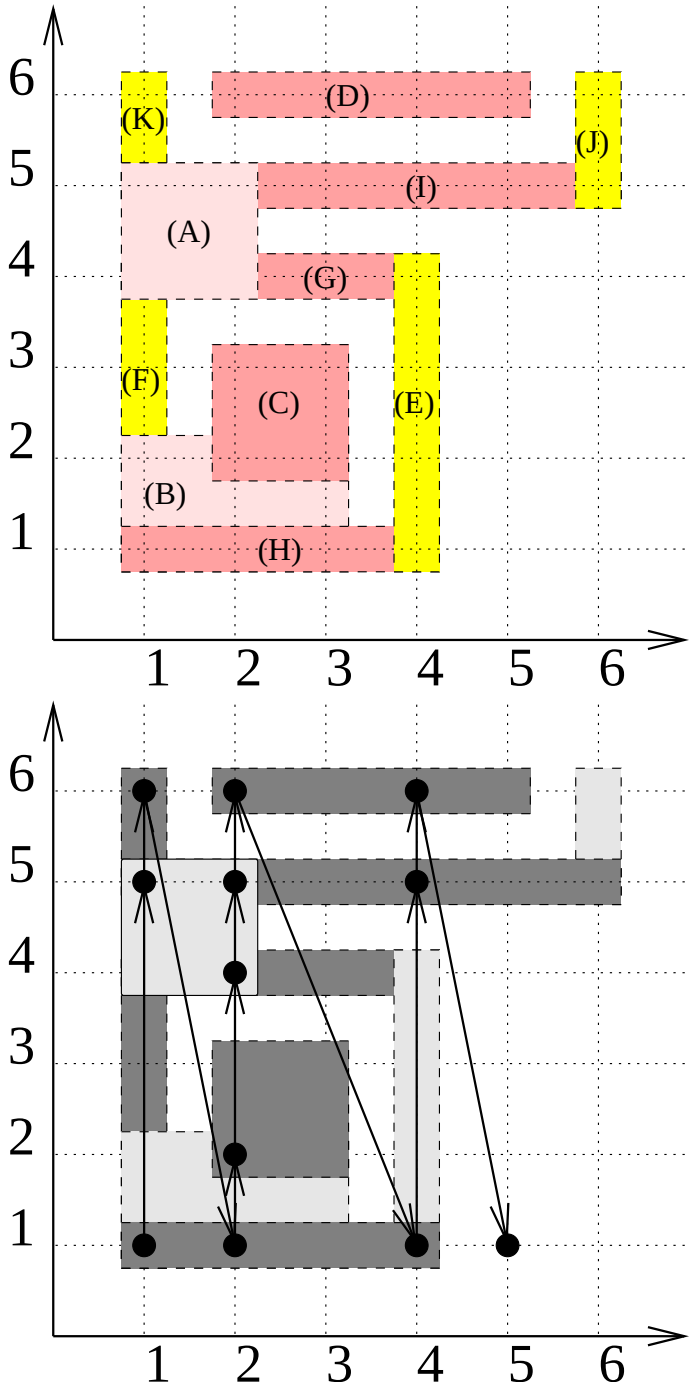

Figure 9: **Top.** Running example: forbidden orthotopes generated by (11), annotated correspondingly with relevant lines of the FOQA procedure shown in Algorithm 1. **Bottom.** Sequence of candidate positions explored by the sweep-based algorithm in order to reach the feasible position $(5, 1)$. The only purpose of using different colors and shadows of grey is to show the borders of the forbidden orthotopes.

Packing-unpacking. We studied the packing-unpacking problem given in Appendix B, where 48 rectangles have to be packed and unpacked in a bin, each rectangle has a time window under which it has to be in the bin as well as a constraint that there be no obstacle at the time of packing and unpacking. This problem instance was modeled as a *geost* constraint over 48 objects, whose origin variables were fixed by greedy assignment in 2800 msec. 4827 FOQA queries were issued. Again, the execution time was dominated by rule compilation; the time spent in greedy assignment disappeared in the noise. If the placement rules are expressed as constraints in conjunction with *geost*, greedy assignment has to be replaced by standard labeling search, which needed 39 backtracks. The execution time was 930 msec.

No-touch square tiling. The problem consists in tiling a square of a given size with several smaller square-shaped tiles, where two tiles must be at least one unit apart in at least one dimension if they are of the same size. The problem was suggested by E. Friedman.[4] Friedman allows the corners of two tiles of the same size to touch; in our formulation, we do not. Multiple tiles of the same shape gave rise to symmetries, broken by lexicographic ordering constraints. We ran instances of sizes 16, 18, 24, 26, 27, 28. The results for finding the first solutions, comparing handling the distance constraints inside vs. in conjunction with *geost*, are shown in Table 4. For this benchmark, handling rules inside *geost* saved less search effort than for the other benchmarks, but did not cause much overhead either. See also Appendix C.

Vessel loading. The problem [26, problem 008] consists in placing on a 16×16 deck ten containers, divided into three classes A, B and C. Containers of classes B and C are constrained to be at least 4 units apart in at least one dimension. All containers can be rotated by 90 degrees. The three classes are:

A Containers of size 6×8, 4×6 and 4×4.

B Containers of size 4×4, 4×4 and 4×6.

C Containers of size 4×8, 4×8, 4×6 and 4×6.

This was modeled as a single *geost* constraint with polymorphic objects and rules imposing 12 pairwise distance constraints. Multiple containers of identical class and size gave rise to symmetries, broken by lexicographic ordering constraints. We ran the original problem as well as an instance with six identical copies of each container (60 in total) to be placed on a 48×32 deck. For the 16×16 instance, we found the first solution after 17 backtracks in some 70 msec, while 164 backtracks were reported in [9]. For the 48×32 instance, the first solution was found after 15 backtracks in some 2300 msec, while 832 backtracks were reported in [9]. The results for the first 100 solutions, comparing handling the distance constraints inside vs. in conjunction with *geost*, are shown in Table 4. See also Appendix D.

Floor planning. We studied several instances from the PhD thesis of B. Medjdoub [10]. These instances have several common features:

- A floor of a given size must be covered by a given number of rooms, i.e. no slack is allowed.
- The area of each room must be inside given bounds.
- The length and width of each room have given lower bounds.
- Each room can be constrained to face the exterior in a given direction. Conjunctions and disjunctions of such constraints also occur.
- A pair of rooms can be constrained to abut, i.e. to meet and have a nonempty intersection on one border. Conjunctions and disjunctions of such constraints also occur.
- The total area occupied by corridors may be subject to minimization.

[4] See `http://www.stetson.edu/~efriedma/mathmagic/0705.html`.

- Multiple rooms with identical restrictions size gave rise to symmetries, broken by lexicographic ordering constraints.

The instances that we studied are listed below, with a summary of their features. The details can be found in [10]. The results for finding the first solutions, comparing handling the topological constraints inside vs. in conjunction with *geost*, are shown in Table 4. See also Appendix E.

maculet1 10 rooms must cover a 12×10 floor. The total area occupied by corridors should be minimized.

maculet2 Similar to **maculet1**, but with extra topological constraints added.

offices 16 rooms must cover a 15×15 floor. No corridor area minimization.

house This is a 3-dimensional instance. 5 rooms must cover a 20×16 area on the first level; 8 rooms must cover the same area on the second level. A staircase connects the two levels. No corridor area minimization.

kindergarten 18 rooms must cover a 28×50 floor. Five of the rooms are constrained to have the same area. No corridor area minimization.

Square Tiling	**backtracks**		**runtime**	
first solution	**rules in**	**rules out**	**rules in**	**rules out**
16	1962	2048	3620	4280
18	786	787	1650	1850
24	3530	3577	9000	11250
26	32	32	120	70
27	2703	2709	7700	7360
28	1882	1882	5170	6580

Vessel Loading	**backtracks**		**runtime**	
first 100 solutions	**rules in**	**rules out**	**rules in**	**rules out**
10 containers on 16×16	3231	4116	1980	3940
60 containers on 48×32	2578	2578	4150	2200

Floor Planning	**backtracks**		**runtime**	
first solution	**rules in**	**rules out**	**rules in**	**rules out**
maculet1	6677	18235	15580	179000
maculet2	17892	75965	88690	101230
offices	7739	12706	136820	22600
house	977	22759	2500	47790
kindergarten	200	29495	1500	43530

Table 4: Runtimes (msec) and backtrack counts for test instances. Side-constraints are handled inside *geost* in columns marked **rules in** resp. posted in conjunction with *geost* in columns marked **rules out**.

6.2 Summary of Results

Due to the small numbers of test instances, the results should be treated with caution. However, the results clearly show that handling rules inside *geost* consistently saves search effort, sometimes by orders of magnitude. The runtime figures are not as consistent: dramatic speedup was observed, but also significant slowdown in one case.

The results reported above for the **rules in** test cases are from a first integration of rules into the sweep-point kernel. No attempts to optimize the implementation have been made, in contrast with the

rules out test cases, where we used built-in constraints, mainly propositional combinations of arithmetic constraints, the implementation of which has matured over many years. So there is scope for improved runtime numbers as the implementation matures. In particular, we have not implemented a rule entailment check, so it's likely the case that significant time is being spent in FOQA procedures for rules that are already entailed, i.e., will never return any forbidden orthotopes.

7 Discussion

Convenience. Rules in our language are often more succinct than the equivalent formulation as separate constraints. The quantifiers provide convenient iteration over sets and their combinations, and every *fol* can be reified. The constraint formulation tends to need more Boolean variables, for the SICStus solver does not allow to reify all constraints. Iteration over sets and their combinations must be expressed with recursion or metacalls.

Generality. Our restriction that object attributes (except shape id and origin) must be ground could be lifted. The rewritten QFPA formulas would simply have more variables per object, and the sweep-based algorithm would deal not with a k- or $k+1$-dimensional *placement* space, but with an m-dimensional *solution* space, where m is the number of possibly nonground attributes per object. In particular, in order to deal with objects whose length in some dimension is a domain variable that occurs in some other constraint, the length and possibly the end-point would have to be expressed as nonground object attributes. Similarly, to treat the time dimension, we would add three nonground object attributes $start$, $duration$, and $completion$, as in [7], to be included in the solution space.

Theoretical properties. It has been shown [4, Proposition 1-2] that the `PKML/Rules2CP` rewriting system is confluent and Noetherian (i.e., terminating). Since our rule language is essentially a subset of `Rules2CP`, the results apply to *geost* rules as well. A size bound on programs generated from `Rules2CP` is also known [4, Proposition 3] and applies to *geost* provided that $\min$, $\max$ and cardinality is not used in the rules, since these operators can cause an exponential (for $\min$ and $\max$) resp. quadratic (for cardinality) [14] blow-up. Consequently, one can certainly construct pathological cases where the rewrite phases and/or runtime representation require huge amounts of memory. Even if, at this time, this has not really been a problem for the instances and rules we have experimented with [5], one way to manage the complexity of the rewrite phases is to apply simplifying rewrites, e.g. Phase 8, as eagerly as possibly. Another way could be to memoize patterns that have already been rewritten. Finally, common subexpression elimination will mitigate this problem.

8 Conclusion

We have presented a global constraint that enforces rules written in a language based on arithmetic and first-order logic to hold among a set of objects. By rewriting the rules to QFPA formulas, we have shown how to compile them first to FS-generators and then to procedures answering queries about candidate points of origins for a given object. Such queries are at the heart of the constraint's sweep-based filtering algorithm. Initial experiments support the feasibility of the approach and that efficient and effective filtering is possible. The approach combines an expressive logic-based rule modeling language for stating business rules with a generic geometrical algorithm for effective filtering.

An open issue is how to infer automatically from a set of rules global necessary conditions for feasibility, for instance how to extract from non-overlapping rules the fact that the available space must be at least the total volume of the objects to place, or how to infer automatically from adjacency rules that the total perimeter of the corridors must be at least the total length of its border intersection with other rooms.

Finally, QFPA is a language in which also many other problems, unrelated to packing and placement, can be stated. In this paper, we have begun to explore a way to compile and run it efficiently.

[5] They involved at most 100 objects.

Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 "Net-WMS". The second author was also partly supported by ANR (CANAR/06-BLAN-0383-02). The data and placement rules used in Appendix A were kindly provided by PSA Peugeot Citroën.

References

[1] N. Beldiceanu and É. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.

[2] N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4):459–489, 2008.

[3] X. Lorca. *Contraintes de partitionnement de graphe*. PhD thesis, Université de Nantes, 2007. In French.

[4] F. Fages and J. Martin. From rules to constraint programs with the `Rules2CP` modelling language. Research Report RR-6495, INRIA Rocquencourt, 2008.

[5] M. Ågren, N. Beldiceanu, M. Carlsson, M. Sbihi, C. Truchet, and S. Zampelli. Six ways of integrating symmetries within non-overlapping constraints. Technical Report T2009-01, Swedish Institute of Computer Science, 2009.

[6] M. Carlsson, N. Beldiceanu, and J. Martin. A geometric constraint over k-dimensional objects and shapes subject to business rules. In P.J. Stuckey, editor, *Proc. CP'2008*, volume 5202 of *LNCS*, pages 220–234. Springer, 2008.

[7] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessière, editor, *Proc. CP'2007*, volume 4741 of *LNCS*, pages 180–194. Springer, 2007.

[8] N. Beldiceanu, M. Carlsson, and E. Poder. New filtering for the *cumulative* constraint in the context of non-overlapping rectangles. In L. Perron and M.A. Trick, editors, *CPAIOR*, volume 5015 of *LNCS*, pages 21–35, Paris, 2008. Springer.

[9] K.N. Brown. Loading supply vessels by forward checking and unenforced guillotine cuts. In *Proc. 17th workshop of the UK Planning and Scheduling Special Interest Group*. University of Huddersfield, 1998.

[10] B. Medjdoub. *Méthode de conception fonctionnelle en architecture: une approche CAO basée sur les contraintes: ARCHiPLAN*. PhD thesis, École Centrale de Paris, 1996. In French.

[11] F. Bacchus and T. Walsh. Propagating logical combinations of constraints. In *IJCAI-05, Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 35–40, 2005.

[12] M. Benedetti, A. Lallouet, and J. Vautard. QCSP made practical by virtue of restricted quantification. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 38–43, 2007.

[13] P. Van Hentenryck and Y. Deville. The *cardinality* operator: a new logical connective in constraint logic programming. In *Int. Conf. on Logic Programming (ICLP'91)*. MIT Press, 1991.

[14] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

[15] V. Ganesh, S. Berezin, and D.L. Hill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Proc. FMCAD'02*, volume 2517 of *LNCS*, pages 171–186. Springer, 2002.

[16] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[17] O. Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In Jean-Charles Régin and Michel Rueher, editors, *Proc. CP-AI-OR 2004*, volume 3011 of *LNCS*, pages 209–224. Springer, 2004.

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

[19] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[20] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[21] B. Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University, 1995.

[22] G. Tack, C. Schulte, and G. Smolka. Generating propagators for finite set constraints. In F. Benhamou, editor, *Proc. CP'2006*, volume 4204 of *LNCS*, pages 575–589. Springer, 2006.

[23] K.C.K. Cheng, J.H.M. Lee, and P.J. Stuckey. Box constraint collections for adhoc constraints. In F. Rossi, editor, *Proc. CP'2003*, volume 2833 of *LNCS*, pages 214–228. Springer, 2003.

[24] W. Harvey and P. J. Stuckey. Constraint representation for propagation. In M. Maher and J.-F. Puget, editors, *Proc. CP'98*, volume 1520 of *LNCS*, pages 235–249. Springer, 1998.

[25] N. Beldiceanu and M. Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 377–391. Springer-Verlag, 2001.

[26] I.P. Gent and T. Walsh. CSP LIB: A benchmark library for constraints. In *Principles and Practice of Constraint Programming*, pages 480–481, 1999. `http://www.csplib.org/`.

[27] H. Simonis and B. O'Sullivan. Search strategies for rectangle packing. In P. J. Stuckey, editor, *Principles and Practice of Constraint Programming (CP'2008)*, volume 5202 of *LNCS*, pages 52–66. Springer-Verlag, 2008.

A Benchmark Placement

This problem instance was provided by a major car manufacturer, involving a $1203 \times 235 \times 239$ container and 9 objects (with *oid* 1-9) with an extra *weight* attribute, subject to the following rules:

inside Each object is placed inside the container (handled by the *geost* kernel).

non_overlap The objects do not pairwise overlap (handled by the *geost* kernel).

gravity Each object is either on the floor or resting on some other object.

stack_weight A heavier object cannot be piled on top of a lighter one.

stack_oversize For any two objects in a pile, the overhang can be at most 10 units.

B Benchmark Packing-Unpacking

This packing-unpacking problem takes the space as well as the time dimensions into account. We have to pack (and unpack) a set of 48 rectangles into a bin. Each rectangle is present within the bin during a given time window and the right hand side of the bin can be used for inserting and deleting rectangles. Beside the fact that all rectangles that are present in the bin at any time instant should not overlap, we also have a visibility constraint, which states that, when a rectangle enters (or leaves) the bin, there should not be any obstacle between the final (initial) position of the rectangle and the right hand side of the bin, assuming that the rectangle is moved horizontally only,

The example illustrates how a packing plan can be obtained for such a packing-unpacking problem from a solution to a *geost* constraint problem. The example uses problem dimensions 1-2 for space and 3 for time. Let $pack(o)$ resp. $unpack(o)$ denote the start resp. completion time of presence in the bin of object o, the duration being given by the associated shifted box. We now introduce the visibility constraint.

Definition 3 *Given a list $OIDs$ of identifiers of objects of the* geost *constraint and an observation place, specified by a* dimension Dim *(1 or 2) and a* direction Dir *(0 or 1), the* $\mathrm{visible}(OIDs, Dim, Dir)$ *constraint holds if, for all objects o mentioned in $OIDs$,* at least one surface of each shifted box associated with o is entirely visible from the specified observation place $\langle Dim, Dir\rangle$ at time $pack(o)$[6] as well as at time $unpack(o) - 1$.[7]

Definition 4 *Consider two distinct objects o and o' of the* $\mathrm{visible}(OIDs, Dim, Dir)$ *constraint (i.e., $o, o' \in OIDs$) as well as an observation place defined by the pair $\langle Dim, Dir\rangle$. The object o* is masked by *the object o' according to the observation place $\langle Dim, Dir\rangle$ if there exist two shifted boxes s and s' respectively associated with o and o' such that conditions* **A**, **B** *and* **C** *all hold:*

A (o, s) *and* (o', s') *intersect in all space and time dimensions except Dim (i.e., (o, s) and (o', s') are in vis-à-vis in dimension Dim).*

B (o, s) *precedes* (o', s') *in dimension Dim and $Dir = 1$, or (o', s') precedes (o, s) in dimension Dim and $Dir = 0$.*

C $pack(o) > pack(o') \vee unpack(o) < unpack(o')$.

The solution is shown in Fig. 10. The four parts of the figure respectively correspond to the successive states of the bin (i.e., we have four time intervals), from left to right:

- Initially, rectangles 1 to 16 enter the bin.

[6] We assume that all objects for which the start time equals $pack(o)$ are transparent. This makes sense since: (1) within the context of pick-up delivery problems all objects loaded (resp. unloaded) at the same place are equivalent; (2) by enforcing the start time to be distinct (for instance by using an $alldifferent$ constraint on the start variables) one can force the objects to be opaque.

[7] Again, we assume that all objects for which the completion time equals $unpack(o)$ are transparent.

- Later on, rectangles 17 to 32 enter the bin. They are placed into the bin in order not to block according to the right hand side of the bin, rectangles 1 to 16 which have to leave earlier.
- Rectangles 1 to 16 leave the container and are replace by rectangles 33 to 48. Again they are placed in order not to block the exit of rectangles 17 to 32.
- After the exit of rectangles 17 to 32, rectangles 33 to 48 are the only rectangles left in the bin.

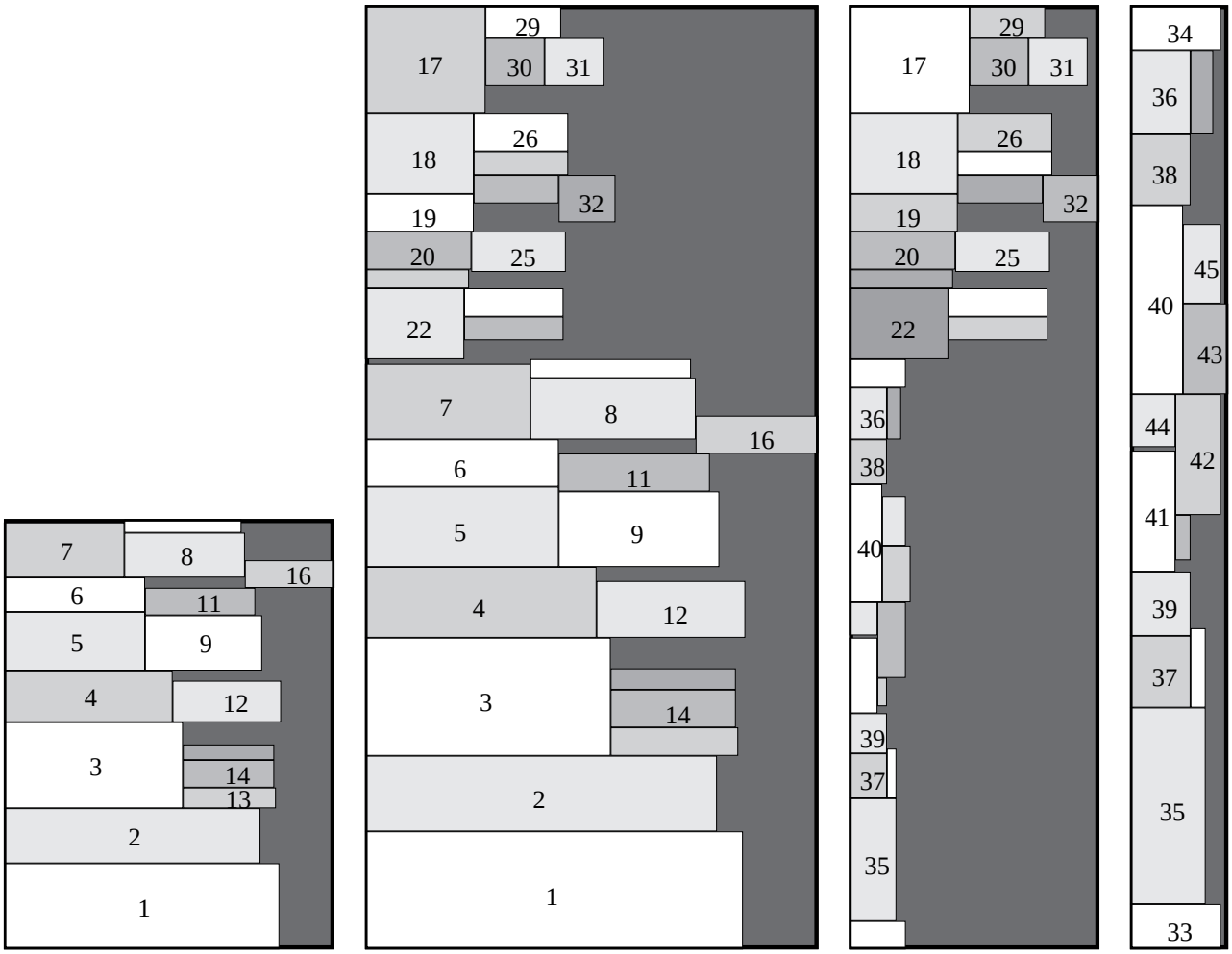

Figure 10: Solution to the packing-unpacking problem.

C Benchmark Square Tiling

The problem consists in tiling a square of a given size with several smaller square-shaped tiles, where two tiles must be at least one unit apart in at least one dimension if they are of the same size. The problem was suggested by E. Friedman.[8] Friedman allows the corners of two tiles of the same size to touch; in our formulation, we do not. Multiple tiles of the same shape gave rise to symmetries, broken by lexicographic ordering, handled inside *geost*. We used the *dual* search strategy [27].

[8] See `http://www.stetson.edu/~efriedma/mathmagic/0705.html`.

D Benchmark Vessels

The problem [26, problem 008] consists in placing on a 16×16 deck ten containers, divided into three classes A, B and C. Containers of classes B and C are constrained to be at least 4 units apart in at least one dimension. All containers can be rotated by 90 degrees. The three classes are:

A Containers of size 6×8, 4×6 and 4×4.

B Containers of size 4×4, 4×4 and 4×6.

C Containers of size 4×8, 4×8, 4×6 and 4×6.

This was modeled as a single *geost* constraint with polymorphic objects and rules imposing 12 pairwise distance constraints. Multiple containers of identical class and size gave rise to symmetries, broken by lexicographic ordering, handled inside *geost*. The search strategy repeatedly computes the lexicographically smallest uncovered deck position, and then decides for each unplaced object whether or not it should be placed there, then deciding its orientation, backtracking over all objects.

E Benchmark Floor Planning

We studied several instances from the PhD thesis of B. Medjdoub [10]. These instances have several common features:

- A floor of a given size must be covered by a given number of rooms, i.e. no slack is allowed.
- The area of each room must be inside given bounds.
- The length and width of each room have given lower bounds.
- Each room can be constrained to face the exterior in a given direction. Conjunctions and disjunctions of such constraints also occur.
- A pair of rooms can be constrained to abut, i.e. to meet and have a nonempty intersection on one border. Conjunctions and disjunctions of such constraints also occur.
- Multiple rooms with identical restrictions size gave rise to symmetries, broken by lexicographic ordering, handled inside *geost*.

It is worth noting that one instance is three-dimensional; it concerns a two-storey building. In two instances, the total area occupied by corridors was subject to minimization. In one instance, five rooms were constrained to have the same area. In four instances, a lower bound on the total perimeter of corridors was used as a necessary condition, posted in conjunction with *geost*. In one instance, the search strategy was fixed variable order and dichotomic value choice. For the other instances, we used the same search strategy as in the Vessels benchmark.

F Benchmark Code

This section shows the verbatim Prolog code used in the performance evaluation.

F.1 Benchmark Placement

```
:-use_module(library(lists)).
:-use_module(library(clpfd)).

main(out, Objects) :-
        statistics(runtime, [T1,_]),
```

```
        variables(Objects, _Container, Sboxes, Vars),
        objects_orthos(Objects, Sboxes, Orthos),
        gravity(Orthos, Orthos),
        stack_weight(Orthos, Orthos),
        stack_oversize(Orthos, Orthos),
        Sboxes = [sbox(0,_,[W,H,L])|_],
        Xmax in 0..W,
        Ymax in 0..H,
        Zmax in 0..L,
        geost(Objects, Sboxes, [bounding_box([0,0,0],[Xmax,Ymax,Zmax])]),
        labeling([enum], Vars),
        statistics(runtime, [T2,_]),
        format('~d msec\n', [T2-T1]).

main(in, Objects) :-
        statistics(runtime, [T1,_]),
        fd_statistics(backtracks, _),
        variables(Objects, _Container, Sboxes, _),
        rule(lib, R0, R2),
        rule(gravity, R2, R4),
        rule(stack_weight, R4, R5),
        rule(stack_oversize, R5, R6),
        rule(wedging, R6, R7),
        R7 = [forall(Box,Objects,
                    gravity(Box,Objects) #/\
                    % wedging(Box,_Container,Objects) #/\
                    true),
              forall(Box1,Objects,
                     forall(Box2,Objects,
                            stack_weight(Box1,Box2) #/\
                            stack_oversize(Box1,Box2) #/\
                            true))
             ],
        Sboxes = [sbox(0,_,[W,H,L])|_],
        Xmax in 0..W,
        Ymax in 0..H,
        Zmax in 0..L,
        geost(Objects, Sboxes,
              [bounding_box([0,0,0],[Xmax,Ymax,Zmax]),
               fixall(1,[object(_,min(1),[min(2),min(3),min(4)])])],
              R0),
        statistics(runtime, [T2,_]),
        fd_statistics(backtracks, B),
        format('~d msec, ~d backtracks\n', [T2-T1,B]).

variables(Boxes, Container, [S1|Ss], Vars) :-
        container(1, _, W, H, L),
        mkobject(0, [0,0,0], [W,H,L], [], Container, S1),
        mkobjects(0, 9, [W,H,L], Boxes, Ss, Origs),
        append(Origs, Vars).

objects_orthos([], _, []).
objects_orthos([object(OID,SID,[X,Y,Z],[weight-K])|Objs], Shapes0,
               [o(OID,K,X,L,Y,W,Z,H)|Os]) :-
        selectchk(sbox(SID,_,[L,W,H]), Shapes0, Shapes),
        objects_orthos(Objs, Shapes, Os).

mkobject(ID, Orig, Dim, Atts,
```

```
         object(ID,ID,Orig,Atts),
         sbox(ID,[0,0,0],Dim)).

mkobjects(N, N, _, [], [], []) :- !.
mkobjects(I, N, UB, [O|Os], [S|Ss], [Origs|Origss]) :-
        J is I+1,
        object(J, _, L0, L1, L2, W),
        upper_bounds(UB, Origs),
        mkobject(J, Origs, [L0,L1,L2], [weight-W], O, S),
        mkobjects(J, N, UB, Os, Ss, Origss).

upper_bounds([], []).
upper_bounds([U|Us], [X|Xs]) :-
        X in 0..U,
        upper_bounds(Us, Xs).

rule(lib) -->
        [(origin(O1,S1,D) ---> O1^x(D)+S1^t(D))],
        [(end(O1,S1,D) ---> O1^x(D)+S1^t(D)+S1^l(D))],
        [(soverlap(O1,O2,S1,S2,D) --->  % sboxes overlap in dim D
          end(O1,S1,D) #> origin(O2,S2,D) #/\
          end(O2,S2,D) #> origin(O1,S1,D))],
        [(oversize(O1,O2,S1,S2,D) ---> % amount by which two sboxes overlap in dim D
          max(max(origin(O1,S1,D),origin(O2,S2,D)) - min(origin(O1,S1,D),origin(O2,S2,D)),
              max(end(O1,S1,D),   end(O2,S2,D)) -    min(end(O1,S1,D),   end(O2,S2,D))))].
rule(gravity) -->
        [(gravity(O1,Os) ---> % O1 is either on the floor on sitting on some other box
          forall(S1,sboxes([O1^sid]),
                 (origin(O1,S1,3) #= 0 #\/
                  exists(O2,Os,
                         O1^oid#\=O2^oid #/\
                         exists(S2,sboxes([O2^sid]),
                                soverlap(O1,O2,S1,S2,1) #/\
                                soverlap(O1,O2,S1,S2,2) #/\
                                origin(O1,S1,3) #= end(O2,S2,3)))))))].
rule(stack_weight) -->
        [(stack_weight(O1,O2) ---> % O1 heavier than O2 => O1 not piled above O2
          O1^weight #> O2^weight #=>
          forall(S1,sboxes([O1^sid]),
                 forall(S2,sboxes([O2^sid]),
                        origin(O1,S1,3) #>= end(O2,S2,3) #=>
                        #\ soverlap(O1,O2,S1,S2,1) #\/
                        #\ soverlap(O1,O2,S1,S2,2))))].
rule(stack_oversize) -->
        [(stack_oversize(O1,O2) ---> % for any two objects in a pile,
                                     % the overhang can be at most 10
          O1^oid#\=O2^oid #=>
          forall(S1,sboxes([O1^sid]),
                 forall(S2,sboxes([O2^sid]),
                        (soverlap(O1,O2,S1,S2,1) #/\
                         soverlap(O1,O2,S1,S2,2)) #=>
                        (oversize(O1,O2,S1,S2,1) #=< 10 #/\
                         oversize(O1,O2,S1,S2,2) #=< 10))))].
rule(wedging) --> % based on Julien's formulation
        [(wedged(O1,S1,Oc,Sc,Os,D) ---> % all four faces of O1 in X/Y must lean
                                        % against the container or against some other box
          (origin(O1,S1,D) #= origin(Oc,Sc,D) #\/
           exists(O2,Os,
```

```
                  O1^oid#\=O2^oid #/\
                  exists(S2,sboxes([O2^sid]),
                         origin(O1,S1,D) #= end(O2,S2,D)))) #/\
            (end(O1,S1,D) #= end(Oc,Sc,D) #\/
             exists(O2,Os,
                  O1^oid#\=O2^oid #/\
                  exists(S2,sboxes([O2^sid]),
                         end(O1,S1,D) #= origin(O2,S2,D)))))],

          [(wedging(O1,Oc,Os) --->
            exists(Sc,sboxes([Oc^sid]),
                   forall(S1,sboxes([O1^sid]),
                          wedged(O1,S1,Oc,Sc,Os,1) #/\
                          wedged(O1,S1,Oc,Sc,Os,2))))].

gravity([], _).
gravity([O1|Os], All) :-
        O1 = o(_,_,_,_,_,_,Z1,_),
        Z1 #= 0 #<=> B,
        selectchk(O1, All, Rest),
        gravity(Rest, O1, Bs),
        sum([B|Bs], #>=, 1),
        gravity(Os, All).

gravity([], _, []).
gravity([O2|Os], O1, [B|Bs]) :-
        O1 = o(_,_,X1,L1,Y1,W1,Z1,_),
        O2 = o(_,_,X2,L2,Y2,W2,Z2,H2),
        overlap1d(X1, L1, X2, L2, B1),
        overlap1d(Y1, W1, Y2, W2, B2),
        Z1 #= Z2+H2 #<=> B3,
        B1 #/\ B2 #/\ B3 #<=> B,
        gravity(Os, O1, Bs).

stack_weight([], _).
stack_weight([O1|Os], All) :-
        stack_weight1(All, O1),
        stack_weight(Os, All).

stack_weight1([], _).
stack_weight1([O2|Os], O1) :-
        O1 = o(_,K1,X1,L1,Y1,W1,Z1,_ ),
        O2 = o(_,K2,X2,L2,Y2,W2,Z2,H2),
        (   K1>K2 ->
            Z1 #>= Z2+H2 #<=> B3,
            overlap1d(X1, L1, X2, L2, B1),
            overlap1d(Y1, W1, Y2, W2, B2),
            B1+B2+B3 #< 3
        ;   true
        ),
        stack_weight1(Os, O1).

stack_oversize([], _).
stack_oversize([O1|Os], All) :-
        stack_oversize1(All, O1),
        stack_oversize(Os, All).

stack_oversize1([], _).
```

```
stack_oversize1([O2|Os], O1) :-
        O1 = o(Id1,_,X1,L1,Y1,W1,_ ,_ ),
        O2 = o(Id2,_,X2,L2,Y2,W2,_ ,_ ),
        (   Id1<Id2 ->
            overlap1d(X1, L1, X2, L2, B1),
            overlap1d(Y1, W1, Y2, W2, B2),
            oversize1d(X1, L1, X2, L2, Size1),
            oversize1d(Y1, W1, Y2, W2, Size2),
            B1 #/\ B2 #=> Size1 #=< 10 #/\ Size2 #=< 10
        ;   true
        ),
        stack_oversize1(Os, O1).

overlap1d(X1, L1, X2, L2, B) :-
        X1+L1 #> X2 #/\ X2+L2 #> X1 #<=> B.

oversize1d(X1, L1, X2, L2, Size) :-
        Size #= max(abs(X1-X2), abs((X1+L1)-(X2+L2))).

% PSA FEASIBLE INSTANCE (all rules posted except wedging_rule)

container(1, tc40, 1203, 235, 239).

object(1, cV8269, 224, 224, 111, 413).
object(2, pF8370, 224, 224, 74, 463).
object(3, cV8371, 224, 224, 74, 842).
object(4, cV8142, 224, 224, 148, 422).
object(5, cV8243, 224, 224, 111, 266).
object(6, cV8258, 224, 224, 111, 321).
object(7, pF8035, 224, 224, 222, 670).
object(8, pF7037, 155, 224, 222, 440).
object(9, cF6120, 112, 224, 148, 325).
```

F.2 Benchmark Packing-Unpacking

```
:- use_module(library(between)).
:- use_module(library(lists)).
:- use_module(library(clpfd)).

main(in, Objects) :-
        statistics(runtime, [T1,_]),
        findall(r(W,H), r(W,H), Rs),
        length(Rs, N),
        gen_geost(0, N, Rs, Objects, Shapes),
        numlist(1, N, AllOIDs),
        order_objects(Objects, Ordered),
        geost(Ordered,
              Shapes,
              [fixall(1,[object(_,min(1),[min(2),min(3),min(4)]),
                         object(_,min(1),[min(2),max(3),min(4)])])],
              [(origin(O, S, D) ---> Oˆx(D)+Sˆt(D)),
               (end(O, S, D) ---> Oˆx(D)+Sˆt(D)+Sˆl(D)),
               % pairwise overlap
               (overlap(O, S, Oi, Si, D) --->
                   end(O, S, D) #> origin(Oi, Si, D) #/\
                   end(Oi, Si, D) #> origin(O, S, D)),
               (visible(OIDs, Dim, Dir) --->
```

```
                    #\ exists(O, objects(OIDs), masked(OIDs, O, Dim, Dir))),
                (masked(OIDs, O, Dim, Dir) --->
                    exists(Oi, objects(OIDs),
                           Oiˆoid #\= Oˆoid #/\ masked_by(O, Oi, Dim, Dir))),
                (masked_by(O, Oi, Dim, Dir) --->
                    exists(S, sboxes([Oˆsid]),
                       exists(Si, sboxes([Oiˆsid]),
                          end(O, S, 3) #> origin(Oi, Si, 3) #/\
                          end(Oi, Si, 3) #> origin(O, S, 3) #/\
                          (Dim #\= 1 #=> overlap(O, S, Oi, Si, 1)) #/\
                          (Dim #\= 2 #=> overlap(O, S, Oi, Si, 2)) #/\
                          (Dir #= 0 #=> origin(O, S, Dim) #>= end(Oi, Si, Dim))  #/\
                          (Dir #= 1 #=> origin(Oi, Si, Dim) #>= end(O, S, Dim))  #/\
                          (origin(O, S, 3) #> origin(Oi, Si, 3) #\/ end(O, S, 3) #< end(Oi, Si,
                % BUSINESS RULES
                visible(AllOIDs, 1, 0)]),
        statistics(runtime, [T2,_]),
        format('~d msec\n', [T2-T1]).

main(out, Objects) :-
        statistics(runtime, [T1,_]),
        fd_statistics(backtracks, _),
        findall(r(W,H), r(W,H), Rs),
        length(Rs, N),
        gen_geost(0, N, Rs, Objects, Shapes),
        objects_orthos(Objects, Shapes, Orthos),
        visible(Orthos, Orthos, 1, 0),
        order_objects(Objects, Ordered),
        geost(Ordered, Shapes),
        label1(Ordered),
        statistics(runtime, [T2,_]),
        fd_statistics(backtracks, B),
        format('~d msec, ~d backtracks\n', [T2-T1,B]).

gen_geost(N, N, [], [], []) :- !.
gen_geost(I, N, [r(W,H)|Rs], [Object|Objects], [Shape|Shapes]) :-
        J is I+1,
        Object = object(J,J,[X,Y,S]),
        Shape = sbox(J,[0,0,0],[W,H,D]),
        LimX is 200-W,
        LimY is 400-H,
        X in 0..LimX,
        Y in 0..LimY,
        (3*I < N -> S=0 ; 3*I < 2*N -> S=50 ; S=110),
        D = 100,
        gen_geost(J, N, Rs, Objects, Shapes).

order_objects(Objects, Ordered) :-
        length(Part1, 16),
        length(Part2, 16),
        length(Part3, 16),
        append(Part2, Part3, Part23),
        append(Part1, Part23, Objects),
        order_objects(Part1, Part2, Ordered, Part3).

order_objects([], []) --> [].
order_objects([X|Xs], [Y|Ys]) --> [X,Y],
        order_objects(Xs, Ys).
```

```
label1([]).
label1([O|Objects]) :-
        O = object(_,_,[X,Y|_]),
        labeling([up], [X,Y]),
        label2(Objects).

label2([]).
label2([O|Objects]) :-
        O = object(_,_,[X,Y|_]),
        labeling([up], [X]),
        labeling([down], [Y]),
        label1(Objects).

objects_orthos([], [], []).
objects_orthos([object(OID,_,[X,Y,Z])|Objs], [sbox(_,_,[L,W,H])|Shapes],
               [o(OID,X,L,Y,W,Z,H)|Os]) :-
        objects_orthos(Objs, Shapes, Os).

visible([], _, _, _).
visible([O1|Os], All, 1, 0) :-
        selectchk(O1, All, Rest),
        not_masked_by_any(Rest, O1, 1, 0),
        visible(Os, All, 1, 0).

not_masked_by_any([], _, _, _).
not_masked_by_any([O2|Os], O1, 1, 0) :-
        O1 = o(_,X1,_ ,Y1,W1,S1,D1),
        O2 = o(_,X2,L2,Y2,W2,S2,D2),
        #\ (S1+D1 #> S2 #/\
            S2+D2 #> S1 #/\
            Y1+W1 #> Y2 #/\
            Y2+W2 #> Y1 #/\
            X1 #>= X2+L2 #/\
            (S1 #> S2 #\/ S1+D1 #< S2+D2)),
        not_masked_by_any(Os, O1, 1, 0).

% rectangles: start/end = 1/100, 50/150, or 110/200

r(160,50).
r(149,32).
r(104,50).
r(98,30).
r(82,34).
r(82,20).
r(70,32).
r(70,26).
r(68,32).
r(68,8).
r(64,16).
r(63,24).
r(54,12).
r(53,16).
r(53,9).
r(52,16).
r(51,46).
r(46,34).
r(46,16).
```

```
r(45,16).
r(44,8).
r(42,30).
r(42,12).
r(42,10).
r(40,17).
r(40,16).
r(40,10).
r(36,12).
r(32,14).
r(25,20).
r(25,20).
r(24,20).
r(24,12).
r(24,12).
r(20,52).
r(16,22).
r(16,19).
r(16,19).
r(16,17).
r(14,50).
r(12,32).
r(12,32).
r(12,24).
r(12,14).
r(10,21).
r(6,22).
r(4,21).
r(4,12).
```

F.3 Benchmark Square Tiling

```
:-use_module(library(between)).
:-use_module(library(lists)).
:-use_module(library(clpfd)).

main :- main(dual1d).

main(Search) :-
        member(Size, [16,18,24,26,27,28]), % 23 and 25 touch diagonally
        member(IO, [out,in]),
        time(top(Size,IO,Search,_)),
        fail.
main(_).


data(16, [1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,6]).
data(18, [1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,5,5,6,6,7,7]).
data(22, [1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,6,6,6,6]).
data(23, [1,1,2,2,3,3,4,5,5,7,11,11,12]).
data(24, [1,1,1,1,2,2,2,2,3,3,4,4,4,4,5,5,6,7,7,8,8,9,9]).
data(25, [1,1,1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,7,13]).
data(26, [1,1,1,1,3,3,4,4,4,4,5,5,5,5,6,6,6,6,11,15]).
data(27, [1,1,1,1,2,2,2,2,3,3,3,3,5,5,5,5,7,7,7,7,8,8,8,8,11]).
data(28, [1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,5,5,5,5,6,12,12,16]).

time(Goal) :-
        portray_clause(Goal),
```

```
        statistics(runtime, [T1,_]),
        call(Goal), !,
        statistics(runtime, [T2,_]),
        fd_statistics(backtracks, Backs),
        format('done in ~d msec ~d backtracks\n', [T2-T1,Backs]).

top(Size, InOut, Search, Objects) :-
        data(Size, Tiles),
        reverse(Tiles, Decreasing),
        objects(Decreasing, Objects, 0, Size),
        sort(Tiles, Unique),
        shapes(Unique, Shapes),
        tag_objects(Objects, Tagged),
        keyclumped(Tagged, Clumped),
        lexes(Clumped, Lexes, []),
        length(Objects, N),
        numlist(1, N, OIDs),
        distance_rules(InOut, OIDs, Clumped, Dis, []),
        geost(Objects, Shapes, [cumulative(true),  % LARGE impact
                                disjunctive(true), % LARGE impact
                                longest_hole(true,-1),
                                visavis_init(true),
                                dynamic_programming(true)| % LARGE impact
                                Lexes], Dis),
        keys_and_values(Clumped, _, Clumps),
        search(Search, Clumps, Size).

objects([], [], _, _).
objects([Tile|Tiles], [object(J,Tile,[X,Y])|Os], I, Size) :-
        UB is Size-Tile,
        domain([X,Y], 0, UB),
        J is I+1,
        objects(Tiles, Os, J, Size).

shapes([], []).
shapes([S|Ss], [sbox(S,[0,0],[S,S])|Sboxes]) :-
        shapes(Ss, Sboxes).

tag_objects([], []).
tag_objects([O|Os], [S-O|SOs]) :-
        O = object(_,S,_),
        tag_objects(Os, SOs).

lexes([]) --> [].
lexes([_-[_]|Tagged]) --> !,
        lexes(Tagged).
lexes([_-Objects|Tagged]) --> [lex(OIDs)],
        {maplist(arg(1), Objects, OIDs)},
        lexes(Tagged).

distance_rules(in, OIDs, _) -->
        [(origin(O1,S1,D) ---> O1^x(D)+S1^t(D)),
         (end(O1,S1,D) ---> O1^x(D)+S1^t(D)+S1^l(D)),
         (proxal(O1,S1,O2,S2,D) --->
             end(O1,S1,D)+1 #> origin(O2,S2,D) #/\
             end(O2,S2,D)+1 #> origin(O1,S1,D)),
         (forall(O1,objects(OIDs),
             forall(S1,sboxes([O1^sid]),
```

```
                forall(O2,objects(OIDs),
                    forall(S2,sboxes([O2^sid]),
                        O1^oid #>= O2^oid #\/
                        O1^sid #\= O2^sid #\/
                        #\ proxal(O1,S1,O2,S2,1) #\/
                        #\ proxal(O1,S1,O2,S2,2))))))].
distance_rules(out, _, Clumped) -->
        {distances(Clumped)}.

distances([]).
distances([_-Objects|Tagged]) :-
        distant(Objects),
        distances(Tagged).

distant([]).
distant([O1|Os]) :-
        distant(Os, O1),
        distant(Os).

distant([], _).
distant([O2|Os], O1) :-
        distant2(O1, O2),
        distant(Os, O1).

distant2(object(_,S,[X1,Y1]), object(_,S,[X2,Y2])) :-
        (   X1-X2 #>= S+1
        #\/ X2-X1 #>= S+1
        #\/ Y1-Y2 #>= S+1
        #\/ Y2-Y1 #>= S+1
        ).


search(primal(W), Clumps, _) :-
        append(Clumps, Objects),
        objects_parts(Objects, Xs, Ys, Sizes),
        primal(W, Xs, Ys, Sizes).
search(dual2d, Clumps, Size) :-
        Size1 is Size-1,
        numlist(0, Size1, List),
        map_product(list2, List, List, Points),
        dual2d(Points, Clumps).
search(dual1d, Clumps, Size) :-
        append(Clumps, Objects),
        objects_parts(Objects, Xs, Ys),
        Size1 is Size-1,
        dual1d(Xs, Ys, Size1).

objects_parts([], [], []).
objects_parts([object(_,S,[X,Y])|Os], [S-X|Xs], [S-Y|Ys]) :-
        objects_parts(Os, Xs, Ys).

objects_parts([], [], [], []).
objects_parts([object(_,S,[X,Y])|Os], [X|Xs], [Y|Ys], [S|Ss]) :-
        objects_parts(Os, Xs, Ys, Ss).

primal(W, Xs, Ys, Sizes) :-
        by_intervals(Xs, Sizes, W),
        labeling([bisect], Xs),
```

```
        by_intervals(Xs, Sizes, W),
        labeling([bisect], Ys).

by_intervals([], _, _).
by_intervals([X|Xs], [N|Ns], W) :-
        by_interval(X, N, W),
        by_intervals(Xs, Ns, W).

by_interval(X, _,   _) :-
        integer(X), !.
by_interval(X, Len, W) :-
        fd_min(X, Min),
        fd_max(X, Max),
        Mid is (Min+Max)>>1,
        (   Max-Min < Len*W -> true
        ;   X in Min..Mid,
            by_interval(X, Len, W)
        ;   Mid1 is Mid+1,
            X in Mid1..Max,
            by_interval(X, Len, W)
        ).

dual2d([], []).
dual2d([P|Points0], Groups1) :-
        select_first(Object, Groups1, Groups2),
        assign2(Object, P),
        filter_points(Points0, Object, Points),
        dual2d(Points, Groups2).

select_first(X, [[X]|R],     R).
select_first(X, [[X|Xs]|R],  [Xs|R]) :- Xs\==[].
select_first(X, [A|L],       [A|R]) :-
        select_first(X, L, R).

assign2(object(_,_,[X,Y]), [U,V]) :-
        clpfd:'$fd_in_interval'(X, U, U, 1),
        clpfd:'$fd_in_interval'(Y, V, V, 0),
        clpfd:'$fd_evaluate_indexical'(RC, Global),
        clpfd:evaluate(RC, Global).

filter_points([], _, []).
filter_points([[Px,Py]|Ps0], R, Ps) :-
        R = object(_,S,[X,Y]),
        Px >= X,
        Px < X+S,
        Py >= Y,
        Py < Y+S, !,
        filter_points(Ps0, R, Ps).
filter_points([P|Ps0], R, [P|Ps]) :-
        filter_points(Ps0, R, Ps).

list2(X, Y, [X,Y]).


dual1d(Xs, Ys, Limit) :-
        dual_lex_labeling(Xs, 0, Limit),
        dual_lex_labeling(Ys, 0, Limit).
```

```
dual_lex_labeling([], _, _) :- !.
dual_lex_labeling(L, I, Limit) :-
        dual_lex_labeling(L, L1, I, Limit, J),
        dual_lex_labeling(L1, J, Limit).

dual_lex_labeling([], [], _, J, J).
dual_lex_labeling([T-X|L1], L2, I, J0, J) :-
        (   integer(X) ->
            dual_lex_labeling(L1, L2, I, J0, J)
        ;   X #= I,
            dual_lex_labeling(L1, L2, I, J0, J)
        ;   X #> I,
            fd_min(X, J1),
            J2 is min(J0,J1),
            skip_tag([T-X|L1], T, L1b, L2, L3),
            dual_lex_labeling(L1b, L3, I, J2, J)
        ).

skip_tag([T-X|L1], T, L1b) --> !, [T-X],
        skip_tag(L1, T, L1b).
skip_tag(L1, _, L1) --> [].
```

F.4 Benchmark Vessels

```
:-use_module(library(between)).
:-use_module(library(lists)).
:-use_module(library(structs)).
:-use_module(library(clpfd)).

main  :-
        time(rules_in(parm(1,8,8,2))),
        time(rules_out(parm(1,8,8,2))),
        time(rules_in(parm(6,24,16,2))),
        time(rules_out(parm(6,24,16,2))).

time(Goal) :-
        portray_clause(Goal),
        new(integer, Counter),
        new(integer, Limit),
        statistics(runtime, [T1,_]),
        findall(0, call(Goal,Counter,Limit), _),
        statistics(runtime, [T2,_]),
        get_contents(Counter, contents, Backs),
        dispose(Counter),
        dispose(Limit),
        format('~d msec ~d backtracks\n', [T2-T1,Backs]).

rules_out(Parm, Counter, Limit) :-
        Parm = parm(_,Width,Height,Sep),
        Height1 is Height+1,
        mkobjects(Parm, Objects, Aset, Bset, Cset,
                  [a-[{34,43},{23,32},{22}],
                   b-[{22},{22},{23,32}],
                   c-[{24,42},{24,42},{23,32},{23,32}]]),
        bsplit(Bset, Bset22, _Bset23),
        csplit(Cset, Cset24, Cset23),
        Shapes = [sbox(34,[0,0],[3,4]),
```

```
                  sbox(43,[0,0],[4,3]),
                  sbox(24,[0,0],[2,4]),
                  sbox(42,[0,0],[4,2]),
                  sbox(23,[0,0],[2,3]),
                  sbox(32,[0,0],[3,2]),
                  sbox(22,[0,0],[2,2])],
        geost([object(0,0,[0,0])|Objects],
              [sbox(0,[Width,0],[1,Height1]),sbox(0,[0,Height],[Width,1])|Shapes],
              [lex(Bset22),lex(Cset23),lex(Cset24)]),
        objects_rectangles(Objects, Rectangles, Tuples),
        table(Tuples, [[34,3,4],[43,4,3],[24,2,4],[42,4,2],[23,2,3],[32,3,2],[22,2,2]]),
        search(Rectangles, Aset, Bset, Cset, Sep, Width, Height, Counter),
        inc(Limit, Nsol),
        (Nsol>=100 -> ! ; true).

rules_in(Parm, Counter, Limit) :-
        Parm = parm(_,Width,Height,Sep),
        Height1 is Height+1,
        mkobjects(Parm, Objects, Aset, Bset, Cset,
                  [a-[{34,43},{23,32},{22}],
                   b-[{22},{22},{23,32}],
                   c-[{24,42},{24,42},{23,32},{23,32}]]),
        bsplit(Bset, Bset22, _Bset23),
        csplit(Cset, Cset24, Cset23),
        basic_rules(Rules, [forall(O1,objects(Bset),
                                   forall(O2,objects(Cset),
                                          distal(O1,O2,Sep)))]),
        Shapes = [sbox(34,[0,0],[3,4]),
                  sbox(43,[0,0],[4,3]),
                  sbox(24,[0,0],[2,4]),
                  sbox(42,[0,0],[4,2]),
                  sbox(23,[0,0],[2,3]),
                  sbox(32,[0,0],[3,2]),
                  sbox(22,[0,0],[2,2])],
        geost([object(0,0,[0,0])|Objects],
              [sbox(0,[Width,0],[1,Height1]),sbox(0,[0,Height],[Width,1])|Shapes],
              [lex(Bset22),lex(Cset23),lex(Cset24)],
              Rules),
        objects_rectangles(Objects, Rectangles, Tuples),
        table(Tuples, [[34,3,4],[43,4,3],[24,2,4],[42,4,2],[23,2,3],[32,3,2],[22,2,2]]),
        search(Rectangles, Aset, Bset, Cset, 0, Width, Height, Counter),
        inc(Limit, Nsol),
        (Nsol>=100 -> ! ; true).

search(Rectangles, Aset, Bset, Cset, Sep, Width, Height, Counter) :-
        maplist(get1(Rectangles), Aset, RAset),
        maplist(get1(Rectangles), Bset, RBset),
        maplist(get1(Rectangles), Cset, RCset),
        (Sep>0 -> distant(RBset, RCset, Sep) ; true),
        asplit(RAset, RAset34, RAset23, RAset22),
        bsplit(RBset, RBset22, RBset23),
        csplit(RCset, RCset24, RCset23),
        dual([RCset24,RCset23,RAset34,RAset22,RAset23,RBset22,RBset23],
             Width, Height, Counter).

mkobjects(parm(Mult,Width,Height,_), Objects0, Aset, Bset, Cset,
          [a-AShapes,b-BShapes,c-CShapes]) :-
        mkobjects1(Mult, c, 0, I, CShapes, Width, Height, Objects0, Objects1),
```

```
        mkobjects1(Mult, a, I, J, AShapes, Width, Height, Objects1, Objects2),
        mkobjects1(Mult, b, J, K, BShapes, Width, Height, Objects2, []),
        numlist(1, I, Cset),
        I1 is I+1,
        numlist(I1, J, Aset),
        J1 is J+1,
        numlist(J1, K, Bset).

mkobjects1(0, _, I, I, _, _, _) --> !.
mkobjects1(N, a, I, J, Sets, W, H) --> !,
        {I1 is I+1},
        {I2 is I+2},
        {I3 is I+3},
        [object(I1,S1,[X1,Y1]),
         object(I2,S2,[X2,Y2]),
         object(I3,S3,[X3,Y3])],
        {Sets = [Sh1,Sh2,Sh3]},
        {S1 in Sh1},
        {S2 in Sh2},
        {S3 in Sh3},
        {domain([X1,X2,X3], 0, W)},
        {domain([Y1,Y2,Y3], 0, H)},
        {M is N-1},
        mkobjects1(M, a, I3, J, Sets, W, H).
mkobjects1(N, b , I, J, Sets, W, H) --> !,
        {I1 is I+1},
        {I2 is I+2},
        {I3 is I+3},
        [object(I1,S1,[X1,Y1]),
         object(I2,S2,[X2,Y2]),
         object(I3,S3,[X3,Y3])],
        {Sets = [Sh1,Sh2,Sh3]},
        {S1 in Sh1},
        {S2 in Sh2},
        {S3 in Sh3},
        {domain([X1,X2,X3], 0, W)},
        {domain([Y1,Y2,Y3], 0, H)},
        {M is N-1},
        mkobjects1(M, b, I3, J, Sets, W, H).
mkobjects1(N, c , I, J, Sets, W, H) --> !,
        {I1 is I+1},
        {I2 is I+2},
        {I3 is I+3},
        {I4 is I+4},
        [object(I1,S1,[X1,Y1]),
         object(I2,S2,[X2,Y2]),
         object(I3,S3,[X3,Y3]),
         object(I4,S4,[X4,Y4])],
        {Sets = [Sh1,Sh2,Sh3,Sh4]},
        {call(S1 in Sh1)},
        {call(S2 in Sh2)},
        {call(S3 in Sh3)},
        {call(S4 in Sh4)},
        {domain([X1,X2,X3,X4], 0, W)},
        {domain([Y1,Y2,Y3,Y4], 0, H)},
        {M is N-1},
        mkobjects1(M, c, I4, J, Sets, W, H).
```

```
asplit([], [], [], []).
asplit([A,B,C|L1], [A|L2], [B|L3], [C|L4]) :-
        asplit(L1, L2, L3, L4).

bsplit([], [], []).
bsplit([A,B,C|L1], [A,B|L2], [C|L3]) :-
        bsplit(L1, L2, L3).

csplit([], [], []).
csplit([A,B,C,D|L1], [A,B|L2], [C,D|L3]) :-
        csplit(L1, L2, L3).

objects_rectangles([], [], []).
objects_rectangles([object(_,S,[X,Y|_])|Os],
                   [r(X,Width,Y,Height)|Rs],
                   [[S,Width,Height]|Ts]) :-
        objects_rectangles(Os, Rs, Ts).

basic_rules -->
        [(origin(O1,S1,D) ---> O1^x(D)+S1^t(D)),
         (end(O1,S1,D) ---> O1^x(D)+S1^t(D)+S1^l(D)),
         (proxal(O1,O2,S1,S2,Sep,D) --->
             end(O1,S1,D)+Sep #> origin(O2,S2,D) #/\
             end(O2,S2,D)+Sep #> origin(O1,S1,D)),
         (distal(O1,O2,Sep) --->
             forall(S1,sboxes([O1^sid]),
                    forall(S2,sboxes([O2^sid]),
                           #\ proxal(O1,O2,S1,S2,Sep,1) #\/
                           #\ proxal(O1,O2,S1,S2,Sep,2))))].

distant(L1, L2, Sep) :-
        maplist(distant1(L2, Sep), L1).

distant1(L2, Sep, R1) :-
        maplist(distant2(R1, Sep), L2).

distant2(r(X1,W1,Y1,H1), Sep, r(X2,W2,Y2,H2)) :-
        (   X1-(X2+W2) #>= Sep
        #\/ X2-(X1+W1) #>= Sep
        #\/ Y1-(Y2+H2) #>= Sep
        #\/ Y2-(Y1+H1) #>= Sep
        ).

dual(Rects, Width, Height, Counter) :-
        W1 is Width-1,
        H1 is Height-1,
        numlist(0, W1, Xs),
        numlist(0, H1, Ys),
        map_product(list2, Xs, Ys, Points),
        dual(Points, Rects, Counter).

dual([], [], _).
dual([P|Points0], Groups1, Counter) :-
        select_first(Rect, Groups1, Groups2),
        assign2(Rect, P),
        filter_points(Points0, Rect, Points),
        dual(Points, Groups2, Counter).
dual(_, _, Counter) :-
```

```
        inc(Counter, _),
        fail.

select_first(X, [[X]|R],     R).
select_first(X, [[X|Xs]|R],  [Xs|R]) :- Xs\==[].
select_first(X, [A|L],       [A|R]) :-
        select_first(X, L, R).

assign2(r(X,W,Y,_), [U,V]) :-
        clpfd:'$fd_in_interval'(X, U, U, 1),
        clpfd:'$fd_in_interval'(Y, V, V, 0),
        clpfd:'$fd_evaluate_indexical'(RC, Global),
        clpfd:evaluate(RC, Global),
        indomain(W).

filter_points([], _, []).
filter_points([[Px,Py]|Ps0], R, Ps) :-
        R = r(X,W,Y,H),
        Px >= X,
        Px < X+W,
        Py >= Y,
        Py < Y+H, !,
        filter_points(Ps0, R, Ps).
filter_points([P|Ps0], R, [P|Ps]) :-
        filter_points(Ps0, R, Ps).

list2(X, Y, [X,Y]).

get1(List, I, Val) :-
        nth1(I, List, Val).
```

F.5 Benchmark Floor Planning

```
:- use_module(library(between)).
:- use_module(library(lists)).
:- use_module(library(avl)).
:- use_module(library(clpfd)).

:- discontiguous
        space/12,
        symmetry/2,
        contour/3,
        adj/4,
        hint/2.

:- dynamic
        space/12,
        symmetry/2,
        contour/3,
        adj/4,
        hint/2.

%% DATA TYPES
%% chamber(Kind, Name, X, L, Y, W, Z, H, A)
%%   Kind ::= floor | room | corridor
%%   Direction ::= s | n | w | e | sw | se | nw | ne

%% FACTS
```

```
%%   space(ID,Kind,Name,Storey,Minlen,Maxlen,Minwidth,Maxwidth,Minheight,Maxheight,Minarea,Max
%%   contour(ID, Name, List of Direction).
%%   adj(ID, Name, List of Name, MinOverlap).
%%   symetry(ID, List of Name).
%%   hint(ID, Formula).

main :-
        main([maculet1-dual,
              maculet2-dual,
              bureaux-dual,
              house-dual,
              kindergarten-primal(bisect)]).

main(List) :-
        member(Key-Search, List),
        member(Model, [out,in]),
        format('Running ~q.\n', [top(Key,Model,Search)]),
        top(Key,Model,Search,_), fail.
main(_).

%% PROBLEM INSTANCE (from Robert Maculet PhD thesis)

space(maculet1, floor   , home     , 1, 12, 12, 10, 10, 1, 1,  120,  120).
space(maculet1, room    , living   , 1,  4,  _,  4,  _, 1, 1,   33,   42).
space(maculet1, corridor, corridor1, 1,  1,  _,  1,  _, 1, 1,    1,   12).
space(maculet1, corridor, corridor2, 1,  1,  _,  1,  _, 1, 1,    1,   12).
space(maculet1, room    , room1    , 1,  3,  _,  3,  _, 1, 1,   11,   15).
space(maculet1, room    , room2    , 1,  3,  _,  3,  _, 1, 1,   11,   15).
space(maculet1, room    , room3    , 1,  3,  _,  3,  _, 1, 1,   11,   15).
space(maculet1, room    , room4    , 1,  3,  _,  3,  _, 1, 1,   15,   20).
space(maculet1, room    , kitchen  , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(maculet1, room    , shower   , 1,  2,  _,  2,  _, 1, 1,    6,    9).
space(maculet1, room    , toilet   , 1,  1,  _,  1,  _, 1, 1,    1,    2).

contour(maculet1, living , [sw]).
contour(maculet1, kitchen, [s,n]).
contour(maculet1, room1  , [s,n]).
contour(maculet1, room2  , [s,n]).
contour(maculet1, room3  , [s,n]).
contour(maculet1, room4  , [s]).

adj(maculet1, living   , [corridor1,corridor2], 1).
adj(maculet1, shower   , [corridor1,corridor2], 1).
adj(maculet1, toilet   , [corridor1,corridor2], 1).
adj(maculet1, room1    , [corridor1,corridor2], 1).
adj(maculet1, room2    , [corridor1,corridor2], 1).
adj(maculet1, room3    , [corridor1,corridor2], 1).
adj(maculet1, room4    , [corridor1,corridor2], 1).
adj(maculet1, kitchen  , [living]             , 1).
adj(maculet1, kitchen  , [shower]             , 1).
adj(maculet1, toilet   , [kitchen,shower]     , 1).
adj(maculet1, corridor1, [corridor2]          , 1).

symmetry(maculet1, [living]).
symmetry(maculet1, [corridor1,corridor2]).
symmetry(maculet1, [room1,room2,room3]).
symmetry(maculet1, [room4]).
symmetry(maculet1, [kitchen]).
```

```
symmetry(maculet1, [shower]).
symmetry(maculet1, [toilet]).

hint(maculet1, 2*(corridor1^length) +
               2*(corridor1^height) +
               2*(corridor2^length) +
               2*(corridor2^height) - 2 #>= 7).
hint(maculet1, minimize(corridor1^area + corridor2^area)).

%% PROBLEM INSTANCE, modification of the above

space(maculet2, floor   , home     , 1, 12, 12, 10, 10, 1, 1,  120,  120).
space(maculet2, room    , living   , 1,  4,  _,  4,  _, 1, 1,   33,   42).
space(maculet2, corridor, corridor1, 1,  1,  _,  1,  _, 1, 1,    1,   12).
space(maculet2, corridor, corridor2, 1,  1,  _,  1,  _, 1, 1,    1,   12).
space(maculet2, room    , room1    , 1,  3,  _,  3,  _, 1, 1,   11,   15).
space(maculet2, room    , room2    , 1,  3,  _,  3,  _, 1, 1,   11,   15).
space(maculet2, room    , room3    , 1,  3,  _,  3,  _, 1, 1,   11,   15).
space(maculet2, room    , room4    , 1,  3,  _,  3,  _, 1, 1,   15,   20).
space(maculet2, room    , kitchen  , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(maculet2, room    , shower   , 1,  2,  _,  2,  _, 1, 1,    4,    9).
space(maculet2, room    , toilet   , 1,  1,  _,  1,  _, 1, 1,    1,    2).

contour(maculet2, living , [sw]).
contour(maculet2, corridor1, [s,n,w,e]).
contour(maculet2, kitchen, [s,n]).
contour(maculet2, room1  , [s,n]).
contour(maculet2, room2  , [s,n]).
contour(maculet2, room3  , [s,n]).
contour(maculet2, room4  , [s]).

adj(maculet2, living   , [corridor1,corridor2], 1).
adj(maculet2, shower   , [corridor1,corridor2], 1).
adj(maculet2, toilet   , [corridor1,corridor2], 1).
adj(maculet2, room1    , [corridor1,corridor2], 1).
adj(maculet2, room2    , [corridor1,corridor2], 1).
adj(maculet2, room3    , [corridor1,corridor2], 1).
adj(maculet2, room4    , [corridor1,corridor2], 1).
adj(maculet2, kitchen  , [corridor1,corridor2], 1).
adj(maculet2, kitchen  , [living]             , 1).
adj(maculet2, kitchen  , [shower]             , 1).
adj(maculet2, toilet   , [kitchen,shower]     , 1).
adj(maculet2, corridor1, [corridor2]          , 1).

symmetry(maculet2, [living]).
symmetry(maculet2, [corridor1]).
symmetry(maculet2, [corridor2]).
symmetry(maculet2, [room1,room2,room3]).
symmetry(maculet2, [room4]).
symmetry(maculet2, [kitchen]).
symmetry(maculet2, [shower]).
symmetry(maculet2, [toilet]).

hint(maculet2, 2*(corridor1^length) +
               2*(corridor1^height) +
               2*(corridor2^length) +
               2*(corridor2^height) - 2 #>= 8).
hint(maculet2, minimize(corridor1^area + corridor2^area)).
```

```
%% PROBLEM INSTANCE: from Medjdoub PhD: Offices

space(bureaux, floor   , home     , 1, 15, 15, 15, 15, 1, 1,  225,  225).
space(bureaux, room    , patio    , 1,  7,  7,  7,  7, 1, 1,   49,   49).
space(bureaux, corridor, corridor1, 1,  1,  _,  1,  _, 1, 1,    1,   30).
space(bureaux, corridor, corridor2, 1,  1,  _,  1,  _, 1, 1,    1,   30).
space(bureaux, room    , room1    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room2    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room3    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room4    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room5    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room6    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room7    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room8    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room9    , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , room10   , 1,  3,  _,  3,  _, 1, 1,    9,   15).
space(bureaux, room    , toilet1  , 1,  2,  _,  2,  _, 1, 1,    6,    9).
space(bureaux, room    , toilet2  , 1,  2,  _,  2,  _, 1, 1,    6,    9).
space(bureaux, room    , reception, 1,  3,  _,  3,  _, 1, 1,    9,    15).

contour(bureaux, reception , [s,n,w,e]). % hard
contour(bureaux, patio     , [se]).      % hard
contour(bureaux, room4     , [nw]). % hint

adj(bureaux, reception, [corridor1,corridor2], 1).
adj(bureaux, room1    , [corridor1,corridor2], 1).
adj(bureaux, room2    , [corridor1,corridor2], 1).
adj(bureaux, room3    , [corridor1,corridor2], 1).
adj(bureaux, room4    , [corridor1,corridor2], 1).
adj(bureaux, room5    , [corridor1,corridor2], 1).
adj(bureaux, room6    , [corridor1,corridor2], 1).
adj(bureaux, room7    , [corridor1,corridor2], 1).
adj(bureaux, room8    , [corridor1,corridor2], 1).
adj(bureaux, room9    , [corridor1,corridor2], 1).
adj(bureaux, room10   , [corridor1,corridor2], 1).
adj(bureaux, corridor1, [corridor2]          , 1).

symmetry(bureaux, [patio]).
symmetry(bureaux, [room1,room2,room3,room4,room5,room6,room7,room8,room9,room10]).
symmetry(bureaux, [toilet1,toilet2]).
symmetry(bureaux, [reception]).
symmetry(bureaux, [corridor1,corridor2]).

hint(bureaux, 2*(corridor1^length) +
              2*(corridor1^height) +
              2*(corridor2^length) +
              2*(corridor2^height) - 2 #>= 11).

%% PROBLEM INSTANCE: from Medjdoub PhD: 2-storey house

space(house, floor   , home     , 1, 20, 20, 16, 16, 2, 2,  640,  640).
space(house, room    , living   , 1,  6,  _,  6,  _, 1, 1,   72,  128).
space(house, room    , kitchen  , 1,  5,  _,  5,  _, 1, 1,   36,   60).
space(house, room    , bathroom1, 1,  4,  _,  4,  _, 1, 1,   16,   36).
space(house, room    , office   , 1,  6,  _,  6,  _, 1, 1,   36,   60).
space(house, corridor, corridor1, 1,  3,  _,  3,  _, 1, 1,    4,   64).
space(house, stairs  , stairs   , 1,  4,  _,  6,  _, 2, 2,   48,   56).
```

```
space(house, room    , room1    , 2,  6,  _,  6,  _, 1, 1,   36,   60).
space(house, room    , room2    , 2,  6,  _,  6,  _, 1, 1,   48,   60).
space(house, room    , room3    , 2,  6,  _,  6,  _, 1, 1,   48,   60).
space(house, room    , room4    , 2,  6,  _,  6,  _, 1, 1,   48,   72).
space(house, room    , bathroom2, 2,  4,  _,  4,  _, 1, 1,   16,   36).
space(house, room    , bathroom3, 2,  4,  _,  4,  _, 1, 1,   16,   36).
space(house, room    , balcony  , 2,  3,  _,  3,  _, 1, 1,   12,   24).
space(house, corridor, corridor2, 2,  3,  _,  3,  _, 1, 1,    4,   64).

contour(house, living    , [s]). % hard
contour(house, kitchen   , [s,n]). % hard
contour(house, balcony   , [s]). % hard
contour(house, office    , [nw]). % hint
contour(house, room1     , [nw]). % hint
contour(house, room2     , [se]). % hint
contour(house, office    , [s,n,w,e]). % weak hint
contour(house, bathroom1 , [s,n,w,e]). % weak hint
contour(house, room1     , [s,n,w,e]). % weak hint
contour(house, room2     , [s,n,w,e]). % weak hint
contour(house, room3     , [s,n,w,e]). % weak hint
contour(house, room4     , [s,n,w,e]). % weak hint
contour(house, bathroom2 , [s,n,w,e]). % weak hint
contour(house, bathroom3 , [s,n,w,e]). % weak hint

adj(house, living   , [corridor1], 2).
adj(house, kitchen  , [corridor1], 2).
adj(house, bathroom1, [corridor1], 2).
adj(house, office   , [corridor1], 2).
adj(house, kitchen  , [living   ], 2).
adj(house, kitchen  , [bathroom1], 2).
adj(house, room1    , [corridor2], 2).
adj(house, room2    , [corridor2], 2).
adj(house, room3    , [corridor2], 2).
adj(house, room4    , [corridor2], 2).
adj(house, bathroom2, [corridor2], 2).
adj(house, bathroom3, [corridor2], 2).
adj(house, room4    , [bathroom2], 2).
adj(house, room4    , [balcony  ], 2).

symmetry(house, [corridor1]).
symmetry(house, [corridor2]).
symmetry(house, [bathroom1]).
symmetry(house, [bathroom2]).
symmetry(house, [bathroom3]).
symmetry(house, [room4]).
symmetry(house, [room1,room2,room3]).

hint(house, 2*(corridor1^length) + 2*(corridor1^height) #>= 8).
hint(house, 2*(corridor2^length) + 2*(corridor2^height) #>= 14).

%% PROBLEM INSTANCE: from Medjdoub PhD: Kindergarten

space(kindergarten, floor   , home     , 1, 28, 28, 50, 50, 1, 1, 1400, 1400).
space(kindergarten, corridor, patio1   , 1, 24, 24,  9,  9, 1, 1,  160,  240).
space(kindergarten, corridor, patio2   , 1,  8,  8, 20, 20, 1, 1,  160,  240).
space(kindergarten, root    , restarea , 1, 10,  _, 10,  _, 1, 1,  140,  160).
space(kindergarten, room    , rhythm   , 1, 28, 28,  9,  9, 1, 1,  140,  280).
space(kindergarten, room    , class1   , 1,  8,  _,  8,  _, 1, 1,   90,  120).
```

```
space(kindergarten, room    , class2   , 1,  8,  _,  8,  _, 1, 1,   90,  120).
space(kindergarten, room    , class3   , 1,  8,  _,  8,  _, 1, 1,   90,  120).
space(kindergarten, room    , class4   , 1,  8,  _,  8,  _, 1, 1,   90,  120).
space(kindergarten, room    , class5   , 1,  8,  _,  8,  _, 1, 1,   90,  120).
space(kindergarten, room    , kitchen  , 1,  4,  _,  4,  _, 1, 1,  25,    35).
space(kindergarten, room    , bathroom , 1,  4,  _,  4,  _, 1, 1,  25,    35).
space(kindergarten, room    , reception, 1,  4,  _,  4,  _, 1, 1,  20,    35).
space(kindergarten, room    , directors, 1,  3,  _,  3,  _, 1, 1,  10,    15).
space(kindergarten, room    , teachers , 1,  3,  _,  3,  _, 1, 1,  10,    30).
space(kindergarten, room    , waitarea , 1,  3,  _,  3,  _, 1, 1,  12,    30).
space(kindergarten, room    , toilet   , 1,  3,  _,  3,  _, 1, 1,  10,    15).
space(kindergarten, room    , rangement, 1,  3,  _,  3,  _, 1, 1,  10,    15).
space(kindergarten, room    , sickbay  , 1,  3,  _,  3,  _, 1, 1,  10,    15).

contour(kindergarten, class1    , [s,n,w,e]). % hard
contour(kindergarten, class2    , [s,n,w,e]). % hard
contour(kindergarten, class3    , [s,n,w,e]). % hard
contour(kindergarten, class4    , [s,n,w,e]). % hard
contour(kindergarten, class5    , [s,n,w,e]). % hard
contour(kindergarten, reception , [s,n,w,e]). % hard

adj(kindergarten, patio1   , [patio2], 1).
adj(kindergarten, class1   , [patio2], 1).
adj(kindergarten, class2   , [patio2], 1).
adj(kindergarten, class3   , [patio2], 1).
adj(kindergarten, class4   , [patio2], 1).
adj(kindergarten, class5   , [patio2], 1).
adj(kindergarten, restarea , [patio2], 1).
adj(kindergarten, bathroom , [patio2], 1).
adj(kindergarten, waitarea , [patio2], 1).
adj(kindergarten, rhythm   , [patio1  ], 1).
adj(kindergarten, rangement, [patio1  ], 1).
adj(kindergarten, sickbay  , [patio1  ], 1).
adj(kindergarten, toilet   , [patio1  ], 1).
adj(kindergarten, waitarea , [reception], 1).

symmetry(kindergarten, [rhythm]).
symmetry(kindergarten, [patio1]).
symmetry(kindergarten, [patio2]).
symmetry(kindergarten, [class1,class2,class3,class4,class5]).
symmetry(kindergarten, [rangement,sickbay,toilet]).

hint(kindergarten, class1^area #= class2^area).
hint(kindergarten, class1^area #= class3^area).
hint(kindergarten, class1^area #= class4^area).
hint(kindergarten, class1^area #= class5^area).

%% SOLVER (main goal to call)

top(ID,Model,Search,Chambers) :-
        statistics(runtime, [T1,_]),
        top1(ID,Model,Search,Chambers), !,
        statistics(runtime, [T2,_]),
        fd_statistics(backtracks, Backs),
        format('done in ~d msec ~d backtracks\n', [T2-T1,Backs]).

top1(ID,Model,Search,Chambers) :-
        Space = space(ID,_,_,_,_,_,_,_,_,_,_,_),
```

```
        findall(Space, Space, Spaces),
        Floor = space(ID, floor, _, _, Length, Length, Width, Width, Height, Height, Ar, Ar),
        selectchk(Floor, Spaces, RoomsCorridors),
        rooms_corridors(RoomsCorridors, Floor, Chambers, Shapes, Areas),
        sum(Areas, #=, Ar),   % does help
        constraints(Model,ID, Chambers, Shapes, Objects, Classes, Length, Width, Height),
        maplist(chamber_object_pairs(Chambers,Objects), Classes, PairClasses),
        findall(Hint, hint(ID,Hint), Hints),
        do_hints(Hints, Chambers),
        label(Search, PairClasses, Length, Width, Height).

do_hints([], _).
do_hints([minimize(SExpr)|Hs], Chambers) :- !,
        parse(SExpr, Expr, Chambers),
        call(Var #= Expr),
        indomain(Var),
        do_hints(Hs, Chambers).
do_hints([SExpr1 #= SExpr2|Hs], Chambers) :- !,
        parse(SExpr1, Expr1, Chambers),
        parse(SExpr2, Expr2, Chambers),
        call(Expr1 #= Expr2),
        do_hints(Hs, Chambers).
do_hints([SExpr1 #>= SExpr2|Hs], Chambers) :- !,
        parse(SExpr1, Expr1, Chambers),
        parse(SExpr2, Expr2, Chambers),
        call(Expr1 #>= Expr2),
        do_hints(Hs, Chambers).
do_hints([_|Hs], Chambers) :-
        do_hints(Hs, Chambers).

parse(X, X, _) :-
        simple(X), !.
parse(X1+X2, Y1+Y2, Chambers) :-
        parse(X1, Y1, Chambers),
        parse(X2, Y2, Chambers).
parse(X1-X2, Y1-Y2, Chambers) :-
        parse(X1, Y1, Chambers),
        parse(X2, Y2, Chambers).
parse(X1*X2, Y1*Y2, Chambers) :-
        parse(X1, Y1, Chambers),
        parse(X2, Y2, Chambers).
parse(Nameˆ Attr, Var, Chambers) :-
        C = chamber(_,Name,_,_,_,_,_,_,_),
        memberchk(C, Chambers),
        parse_attr(Attr, C, Var).

parse_attr(length, chamber(_,_,_,L,_,_,_,_,_), L).
parse_attr(width,  chamber(_,_,_,_,_,W,_,_,_), W).
parse_attr(height, chamber(_,_,_,_,_,_,_,H,_), H).
parse_attr(area,   chamber(_,_,_,_,_,_,_,_,A), A).

chamber_object_pairs(Chambers, Objects, Class, Pairs) :-
        maplist(chamber_object_pair(Chambers,Objects), Class, Pairs).

chamber_object_pair(Chambers, Objects, Name, C-O) :-
        C = chamber(_,Name,_,_,_,_,_,_,_),
        nth0(I, Chambers, C),
        nth0(I, Objects, O), !.
```

```
rooms_corridors([], _, [], [], []).
rooms_corridors([RC|RestRC], Floor, [O|RestO], [Shs|Shss], [A|As]) :-
        Floor = space(_,_,_,_,Length,Length,Width,Width,Height,Height,Area,Area),
        RC = space(_,KIND,NAME,Z,MINL,MAXL,MINW,MAXW,MINH,MAXH,MINS,MAXS),
        O = chamber(KIND,NAME,X,L,Y,W,Z,H,A),
        L in 1..Length,
        L #>= MINL,
        L #=< MAXL,
        W in 1..Width,
        W #>= MINW,
        W #=< MAXW,
        H in 1..Height,
        H #>= MINH,
        H #=< MAXH,
        A in 1..Area,
        A #>= MINS,
        A #=< MAXS,
        A #= L*W*H,
        X in 1..Length,
        Y in 1..Width,
        findall([L,W,H], labeling([],[L,W,H]), Shs),
        rooms_corridors(RestRC, Floor, RestO, Shss, As).

constraints(in, ID, Chambers, Shapes, Objects, Classes, Length, Width, Height) :-
        Length1 is Length+1,
        Width1 is Width+1,
        Height1 is Height+1,
        append(Shapes, L1),
        sort(L1, L2),
        shapes_sboxes(L2, Sboxes, Extension, 0),
        chambers_objects(Chambers, Shapes, Objects, Tuples, Sboxes, 0),
        table(Tuples, Extension),
        names_and_oids(Chambers, Objects, N2Olist),
        list_to_avl(N2Olist, N2O),
        macros(Rules, Rules0),
        contour_rules(N2Olist, ID, Length1, Width1, Rules0, Rules1),
        adjacency_rules(N2Olist, ID, N2O, Rules1, Rules2),
        (   hint(ID, visavis(Cond)) ->
            visavis_lib(Rules2, [Cond1]),
            visavis_convert(Cond, Cond1, N2O)
        ;   Rules2 = []
        ),
        symmetries(ID, Chambers, Classes, N2O, Lexes),
        Volume is Length*Width*Height,
        geost(Objects, Sboxes,
              [volume(Volume),
               cumulative(true),
               bounding_box([1,1,1],[Length1,Width1,Height1])|Lexes],
              Rules).
constraints(out, ID, Chambers, Shapes, Objects, Classes, Length, Width, Height) :-
        Length1 is Length+1,
        Width1 is Width+1,
        Height1 is Height+1,
        gen_contour(ID, Chambers, Length1, Width1),
        gen_adj(ID, Chambers),
        append(Shapes, L1),
        sort(L1, L2),
```

```
        shapes_sboxes(L2, Sboxes, Extension, 0),
        chambers_objects(Chambers, Shapes, Objects, Tuples, Sboxes, 0),
        table(Tuples, Extension),
        names_and_oids(Chambers, Objects, N2Olist),
        list_to_avl(N2Olist, N2O),
        symmetries(ID, Chambers, Classes, N2O, Lexes),
        Volume is Length*Width*Height,
        geost(Objects, Sboxes,
              [volume(Volume),
               cumulative(true),
               bounding_box([1,1,1],[Length1,Width1,Height1])|Lexes]).

chambers_objects([], [], [], [], _, _).
chambers_objects([chamber(_,_,X,L,Y,W,Z,H,_)|Cs],
                 [Shs|Shss],
                 [object(J,Sid,[X,Y,Z])|Os],
                 [[Sid,L,W,H]|Tuples], Sboxes, I) :-
        J is I+1,
        maplist(getsid(Sboxes), Shs, Sids),
        list_to_fdset(Sids, Fdset),
        Sid in_set Fdset,
        chambers_objects(Cs, Shss, Os, Tuples, Sboxes, J).

getsid(Sboxes, Sh, Sid) :-
        memberchk(sbox(Sid,_,Sh), Sboxes).

names_and_oids([], [], []).
names_and_oids([chamber(_,Name,_,_,_,_,_,_,_)|Cs], [object(OID,_,_)|Os], [Name-OID|NOs]) :-
        names_and_oids(Cs, Os, NOs).

macros -->
        [(origin(O1,S1,D) ---> O1^x(D)+S1^t(D)),
         (end(O1,S1,D) ---> O1^x(D)+S1^t(D)+S1^l(D)),
         (overlap(O1,S1,O2,S2,I,D) --->
             end(O1,S1,D) #>= origin(O2,S2,D)+I #/\
             end(O2,S2,D) #>= origin(O1,S1,D)+I),
         (qadjacent(O1,S1,O2,S2,I) --->
             ((origin(O1,S1,1) #= end(O2,S2,1) #\/ origin(O2,S2,1) #= end(O1,S1,1)) #/\
              overlap(O1,S1,O2,S2,I,2)) #\/
             ((origin(O1,S1,2) #= end(O2,S2,2) #\/ origin(O2,S2,2) #= end(O1,S1,2)) #/\
              overlap(O1,S1,O2,S2,I,1))),
         (adjacent(OID1,OID2,I) --->
             forall(O1,objects([OID1]), forall(S1,sboxes([O1^sid]),
                 forall(O2,objects([OID2]), forall(S2,sboxes([O2^sid]),
                     qadjacent(O1,S1,O2,S2,I))))))].

contour_rules([], _, _, _) --> [].
contour_rules([Name-OID|N2Olist], ID, L, H) -->
        {findall(Cont, contour(ID,Name,Cont), Conts)},
        contour_rules1(Conts, OID, L, H),
        contour_rules(N2Olist, ID, L, H).

contour_rules1([], _, _, _) --> [].
contour_rules1([Cont|Conts], OID, L, H) --> [Rule],
        {contour_rule(Cont, OID, Rule, L, H)},
        contour_rules1(Conts, OID, L, H).

contour_rule([], _, false, _, _).
```

```
contour_rule([Ori|Oris], OID, R #\/ Rs, L, H) :-
        contour_rule(Ori, OID, R, L, H),
        contour_rule(Oris, OID, Rs, L, H).

contour_rule(n, OID, Rule, _, H) :-
        Rule = (forall(O,objects([OID]), forall(S,sboxes([O^sid]), end(O,S,2)#=H))).
contour_rule(s, OID, Rule, _, _) :-
        Rule = (forall(O,objects([OID]), forall(S,sboxes([O^sid]), origin(O,S,2)#=1)))
contour_rule(e, OID, Rule, L, _) :-
        Rule = (forall(O,objects([OID]), forall(S,sboxes([O^sid]), end(O,S,1)#=L))).
contour_rule(w, OID, Rule, _, _) :-
        Rule = (forall(O,objects([OID]), forall(S,sboxes([O^sid]), origin(O,S,1)#=1))).
contour_rule(nw, OID, R1 #/\ R2, L, H) :-
        contour_rule(n, OID, R1, L, H),
        contour_rule(w, OID, R2, L, H).
contour_rule(ne, OID, R1 #/\ R2, L, H) :-
        contour_rule(n, OID, R1, L, H),
        contour_rule(e, OID, R2, L, H).
contour_rule(sw, OID, R1 #/\ R2, L, H) :-
        contour_rule(s, OID, R1, L, H),
        contour_rule(w, OID, R2, L, H).
contour_rule(se, OID, R1 #/\ R2, L, H) :-
        contour_rule(s, OID, R1, L, H),
        contour_rule(e, OID, R2, L, H).

adjacency_rules([], _, _) --> [].
adjacency_rules([Name-OID|N2Olist], ID, N2O) -->
        {findall(Nbs-Atleast, adj(ID,Name,Nbs,Atleast), Nbss)},
        {Nbss \== []}, !,
        {avl_fetch(Name, N2O, OID)},
        adjacency_conj(Nbss, OID, N2O),
        adjacency_rules(N2Olist, ID, N2O).
adjacency_rules([_|N2Olist], ID, N2O) -->
        adjacency_rules(N2Olist, ID, N2O).

adjacency_conj([], _, _) --> [].
adjacency_conj([Nbs-Atleast|Nbss], OID, N2O) --> [Rule],
        {adjacency_rule(Nbs, Atleast, OID, N2O, Rule)},
        adjacency_conj(Nbss, OID, N2O).

adjacency_rule([], _, _, _, false).
adjacency_rule([Nb|Oris], Atleast, OID1, N2O, adjacent(OID1,OID2,Atleast) #\/ Rs) :-
        avl_fetch(Nb, N2O, OID2),
        adjacency_rule(Oris, Atleast, OID1, N2O, Rs).

visavis_lib -->
        [(common(O1,S1,O2,S2,D) --->
          min(end(O2,S2,D)-origin(O1,S1,D),end(O1,S1,D)-origin(O2,S2,D))),
         (diff(O1,S1,O2,S2,D) --->
          max(end(O1,S1,D)-origin(O2,S2,D),origin(O2,S2,D)-end(O1,S1,D))),
         (visavis(O1,S1,O2,S2) --->
          max(0,common(O1,S1,O2,X2,1) - 1000*diff(O1,S1,O2,S2,2)) +
          max(0,common(O1,S1,O2,X2,1) - 1000*diff(O2,S2,O1,S1,2)) +
          max(0,common(O1,S1,O2,X2,2) - 1000*diff(O1,S1,O2,S2,1)) +
          max(0,common(O1,S1,O2,X2,2) - 1000*diff(O2,S2,O1,S1,1)))].

visavis_convert(Rule1, Rule2, N2Oid) :-
        avl_to_list(N2Oid, N2Oidlist),
```

```
        quantify(N2Oidlist, N2Olist, N2Slist, Rule2, Rule3),
        ord_list_to_avl(N2Olist, N2O),
        ord_list_to_avl(N2Slist, N2S),
        visavis_parse(Rule1, Rule3, N2O, N2S).

visavis_parse(X#>=Y, X1#>=Y1, N2O, N2S) :- !,
        visavis_parse(X, X1, N2O, N2S),
        visavis_parse(Y, Y1, N2O, N2S).
visavis_parse(X+Y, X1+Y1, N2O, N2S) :- !,
        visavis_parse(X, X1, N2O, N2S),
        visavis_parse(Y, Y1, N2O, N2S).
visavis_parse(X*Y, X1*Y1, N2O, N2S) :- !,
        visavis_parse(X, X1, N2O, N2S),
        visavis_parse(Y, Y1, N2O, N2S).
visavis_parse(X^length, S^l(1), _, N2S) :- !,
        avl_fetch(X, N2S, S).
visavis_parse(X^height, S^l(2), _, N2S) :- !,
        avl_fetch(X, N2S, S).
visavis_parse(visavis(N1,N2), visavis(O1,S1,O2,S2), N2O, N2S) :- !,
        avl_fetch(N1, N2O, O1),
        avl_fetch(N2, N2O, O2),
        avl_fetch(N1, N2S, S1),
        avl_fetch(N2, N2S, S2).
visavis_parse(X, X, _, _).

quantify([], [], [], R, R).
quantify([Name-Oid|L1], [Name-O|L2], [Name-S|L3],
         exists(O,objects([Oid]),exists(S,sboxes([O^sid]),R0)), R) :-
        quantify(L1, L2, L3, R0, R).

shapes_sboxes([], [], [], _).
shapes_sboxes([LWH|LWHs], [sbox(J,[0,0,0],LWH)|Sboxes], [[J|LWH]|Ext], I) :-
        J is I+1,
        shapes_sboxes(LWHs, Sboxes, Ext, J).

gen_contour(ID, Chambers, Length, Width) :-
        findall(Name-Cont, contour(ID,Name,Cont), L),
        gen_contour1(L, Chambers, Length, Width).

gen_contour1([], _, _, _).
gen_contour1([Name-Contour|R], Chambers, Length, Width) :-
        Chamber = chamber(_,Name,_,_,_,_,_,_,_),
        memberchk(Chamber, Chambers),
        gen_contour2(Contour, Chamber, Length, Width, DISJ),
        call(DISJ),
        gen_contour1(R, Chambers, Length, Width).

gen_contour2([], _, _, _, 0).
gen_contour2([n|R], OBJ, Length, Width, Y+W#=Width #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,_,_,Y,W,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([s|R], OBJ, Length, Width, Y#=1 #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,_,_,Y,_,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([e|R], OBJ, Length, Width, X+L#=Length #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,X,L,_,_,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([w|R], OBJ, Length, Width, X#=1 #\/ REST_DISJ) :- !,
```

```
        OBJ = chamber(_,_,X,_,_,_,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([nw|R], OBJ, Length, Width, (X#=1 #/\ Y+W#=Width) #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,X,_,Y,W,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([ne|R], OBJ, Length, Width, (X+L#=Length #/\ Y+W#=Width) #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,X,L,Y,W,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([se|R], OBJ, Length, Width, (X+L#=Length #/\ Y#=1) #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,X,L,Y,_,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).
gen_contour2([sw|R], OBJ, Length, Width, (X#=1 #/\ Y#=1) #\/ REST_DISJ) :- !,
        OBJ = chamber(_,_,X,_,Y,_,_,_,_),
        gen_contour2(R, OBJ, Length, Width, REST_DISJ).


gen_adj(ID, Chambers) :-
        findall(adj(Name,Adjacencies,I), adj(ID,Name,Adjacencies,I), L),
        gen_adj1(L, Chambers).

gen_adj1([], _).
gen_adj1([adj(Name,Adjacencies,I)|R], Chambers) :-
        Chamber = chamber(_,Name,_,_,_,_,_,_,_),
        memberchk(Chamber, Chambers),
        gen_adj2(Adjacencies, I, Chamber, Chambers, DISJ),
        call(DISJ),
        gen_adj1(R, Chambers).

gen_adj2([], _, _, _, 0).
gen_adj2([Name1|R], I, Chamber2, Chambers,
         (X1+L1#=X2 #/\ Y1+W1#>=Y2+I #/\ Y2+W2#>=Y1+I) #\/
         (X2+L2#=X1 #/\ Y1+W1#>=Y2+I #/\ Y2+W2#>=Y1+I) #\/
         (Y1+W1#=Y2 #/\ X1+L1#>=X2+I #/\ X2+L2#>=X1+I) #\/
         (Y2+W2#=Y1 #/\ X1+L1#>=X2+I #/\ X2+L2#>=X1+I) #\/
         REST_DISJ) :-
        Chamber1 = chamber(_,Name1,X1,L1,Y1,W1,_,_,_),
        Chamber2 = chamber(_,    _,X2,L2,Y2,W2,_,_,_),
        memberchk(Chamber1, Chambers),
        gen_adj2(R, I, Chamber2, Chambers, REST_DISJ).

symmetries(ID, Chambers, AllClasses, N2O, Lexes) :-
        findall(Class, symmetry(ID,Class), Classes),
        maplist(arg(2), Chambers, Names),
        symmetry_classes(Classes, Names, AllClasses, []),
        nonsingletons(Classes, Classes1),
        maplist(make_lex(N2O), Classes1, Lexes).

nonsingletons([], []).
nonsingletons([X|Xs], [X|Ys]) :-
        X = [_,_|_], !,
        nonsingletons(Xs, Ys).
nonsingletons([_|Xs], Ys) :-
        nonsingletons(Xs, Ys).

symmetry_classes([], Names) -->
        symmetry_classes(Names).
symmetry_classes([Class|Classes], Names0) --> [Class],
        {filter_names(Class, Names0, Names)},
```

```
        symmetry_classes(Classes, Names).

symmetry_classes([]) --> [].
symmetry_classes([Name|Names]) --> [[Name]],
        symmetry_classes(Names).

filter_names([], Ns, Ns).
filter_names([X|Xs], Ns0, Ns) :-
        selectchk(X, Ns0, Ns1),
        filter_names(Xs, Ns1, Ns).

make_lex(N2O, Names, lex(OIDs)) :-
        maplist(fetch_avl(N2O), Names, OIDs).

fetch_avl(AVL, Key, Val) :-
        avl_fetch(Key, AVL, Val).

lex_chain_class(Chambers, Class) :-
        maplist(getxy(Chambers), Class, Chain),
        lex_chain(Chain, [op(#<)]).

getxy(Chambers, Name, [X,Y]) :-
        memberchk(chamber(_,Name,X,_,Y,_,_,_,_), Chambers).

label(primal(P), Classes, _, _, _) :- % P ::= enum | step | bisect
        append(Classes, Pairs),
        pairs_vars(Pairs, Vars, []),
        labeling([P], Vars).
label(dual, Classes, Length, Width, Height) :-
        numlist(1, Length, Xs),
        numlist(1, Width, Ys),
        numlist(1, Height, Zs),
        map_product(cons, Zs, [[]], Points0),
        map_product(cons, Ys, Points0, Points1),
        map_product(cons, Xs, Points1, Points),
        dual(Points, Classes).

pairs_vars([]) --> [].
pairs_vars([chamber(_,_,X,_,Y,_,_,_,A)-object(_,S,_)|Rs]) --> [A,S,X,Y],
        pairs_vars(Rs).

dual([], []).
dual([P|Points0], Groups1) :-
        Ch = chamber(_,_,_,L,_,W,_,H,A),
        select_first(Ch-Obj, Groups1, Groups2),
        assign2(Obj, P, Sid),
        labeling([bisect], [A,Sid]),
        filter_points(Points0, Obj-[L,W,H], Points),
        dual(Points, Groups2).

select_first(X, [[X]|R],     R).
select_first(X, [[X|Xs]|R],  [Xs|R]) :- Xs\==[].
select_first(X, [A|L],       [A|R]) :-
        select_first(X, L, R).

assign2(object(_,Sid,[X,Y,_]), [U,V,_], Sid) :-
        clpfd:'$fd_in_interval'(X, U, U, 1),
        clpfd:'$fd_in_interval'(Y, V, V, 0),
```

```
        clpfd:'$fd_evaluate_indexical'(RC, Global),
        clpfd:evaluate(RC, Global).

filter_points([], _, []).
filter_points([[Px,Py,Pz]|Ps0], O, Ps) :-
        O = object(_,_,[X,Y,Z])-[L,W,H],
        Px >= X,
        Px < X+L,
        Py >= Y,
        Py < Y+W,
        Pz >= Z,
        Pz < Z+H, !,
        filter_points(Ps0, O, Ps).
filter_points([P|Ps0], O, [P|Ps]) :-
        filter_points(Ps0, O, Ps).
```