

Search as Constraint Satisfaction

Thierry Martinez, François Fages, Philippe Morignot, and Sylvain Soliman

Inria Paris-Rocquencourt, France

Abstract. Constraint programming is traditionally viewed as the combination of two components: a constraint model and a search procedure. In this paper we show that tree search procedures can be fully internalized in the constraint model with a fixed enumeration strategy. This approach has several advantages: 1) it makes search strategies declarative, and modeled as constraint satisfaction problems; 2) it makes it possible to express search strategies in existing front-end modeling languages supporting reified constraints without any extension; 3) it opens up constraint propagation algorithms to search constraints and to the implementation of novel search procedures based on constraint propagation. We illustrate this approach with the modeling of a variety of search procedures, including dynamic symmetry breaking procedures and limited discrepancy search, as constraint satisfaction problems. We show that this generality does not come with a significant overhead, and can in fact exhibit exponential speedups over procedural implementations, thanks to the propagation of the search constraints.

1 Introduction

Constraint programming is traditionally presented as the combination of two components: a constraint model and a search procedure [17]. Front-end modeling languages are designed for solving problems using constraint programming solvers, thus either rely on a fixed strategy (e.g. Essence [6]), or contain special features for specifying the search strategy for the constraint solvers (e.g. Zinc [12]). The modeling language Zinc, and its implementation MiniZinc¹, succeeded in becoming a common input format across many solvers in the Constraint Programming community. In Zinc, the search procedure is specified through special annotations that are dedicated to the constraint solver [14] and ignored by the other solvers.

In this paper, we show that a completely different approach for specifying search is possible, by internalizing the search procedure in the constraint model with a fixed enumeration strategy. In principle, transforming search procedures into constraint satisfaction problems presents several advantages:

1. it makes search strategies declarative, and modeled as constraint satisfaction problems;

¹ <http://www.minizinc.org/>

2. it makes it possible to express search strategies in existing front-end modeling languages without any extension;
3. it opens up constraint propagation algorithms to search constraints and to the implementation of novel search procedures based on constraint propagation.

The idea of this transformation is to associate to each choice point a reified constraint with an auxiliary model variable for representing that choice (e.g. value enumeration, domain splitting or any constraint). The search heuristic can then be specified simply by the enumeration strategy for the choice variables. This approach is not limited to static search procedures in which all choice points are precisely known statically, but can accommodate dynamic search strategies, such as dichotomic or interval splitting search [16] for example. In constraint programming, dynamic search procedures rely on the values of indexicals (domain size, minimum value, etc.). They are expressed in the framework presented here by extending the enumeration strategy with annotations that assign the values of indexicals to auxiliary model variables. Static search procedures do not rely on the values of indexicals and their encoding do not need any specific support on the solver-side. The encoding of dynamic search procedures can be run through simple additions in the solvers for providing the capability to query the values of indexicals.

In this paper, to make concrete the presentation of the transformation, we consider the Zinc modeling language and introduce ClpZinc², a language extending Zinc with the ability to describe new relations by Horn clauses. The choice of CLP as a specification language for search procedures is guided by CLP being the smallest language with the addition of constraint to the store as primitive and closed by conjunction and disjunction (for expressing choices), and with a general form of recursion. Given a constraint system \mathcal{X} (e.g. finite domains) and the Herbrand constraint system \mathcal{H} , we describe a partial evaluation procedure to transform any terminating $\text{CLP}(\mathcal{X} + \mathcal{H})$ goal to an and/or tree with constraints over \mathcal{X} .

In Section 2, we introduce the ClpZinc language and describe the transformation of search procedures from $\text{CLP}(\mathcal{X} + \mathcal{H})$ to and/or trees over \mathcal{X} with some tree traversal. In Section 3, we describe the transformation from those and/or trees to their internalization in $\text{CSP}(\mathcal{X})$, *i.e.*, into Zinc models with a back-end solver for \mathcal{X} . In the subsequent sections, we evaluate this approach on benchmarks of models with specific search strategies, namely: Korf’s Square Packing problem in Section 4, limited discrepancy search in Section 5.1 and symmetry breaking during search in Section 5.2. In Section 6, we show how it is possible to go beyond tree search procedures by using a simple mechanism of annotations for global store, and specify optimization procedures such as Branch-and-Bound. In Section 7, we conclude on some new perspectives.

² The Clp2Zinc compiler that transforms ClpZinc models into MiniZinc is available for download, together with patches for Choco, JaCoP, SICStus, Gecode and or-tools: <http://lifeware.inria.fr/~tmartine/clp2zinc/>

2 Extending Zinc with CLP Clauses

2.1 The Language ClpZinc

We propose to use Constraint Logic Programming (CLP) clauses to specify search strategies in Zinc. More precisely, given a constraint system \mathcal{X} (e.g. finite domain constraints) and a CSP model with constraints in \mathcal{X} , we consider search procedures that are expressible as the traversal of an and/or tree with constraints over \mathcal{X} , *i.e.* an and/or tree where every leaf is either labeled by a constraint in \mathcal{X} or, for dynamic search procedures, labeled by a query to indexicals. In addition, we consider the Prolog primitive constraint system, \mathcal{H} , *i.e.* Herbrand terms with unification. The choice of Herbrand terms for representing Zinc data structures makes the language look familiar to Prolog users. Similarly, we fix the CLP strategy as depth-first and left-to-right.

The language ClpZinc is an extension of Zinc where the item `solve satisfy`; in models is replaced by a CLP goal of the form “`:- goal.`”, and where user-defined predicates are defined by CLP clauses of the form “`p(t1,...,tn) :- goal.`”.

Example 1 (Labeling). The following ClpZinc model implements the search strategy that enumerates all possible values for a given variable in ascending order.

```
var 0..5: x;
constraint x * x = x + x;

labeling(X, Min, Max) :-
  Min <= Max, (X = Min ; labeling(X, Min + 1, Max)).

:- labeling(x, 0, 5).
output [show(x)];
```

As shown in the following section, this ClpZinc model for the given goal of labeling `x` between 0 and 5, can be expanded to the following MiniZinc model:

```
var 0..5: x;
constraint x * x = x + x;
var 0..5: X1;
constraint X1 = 0 -> x = 1;
constraint X1 = 1 -> x = 3;
constraint X1 = 2 -> x = 5;
constraint X1 = 3 -> x = 4;
constraint X1 = 4 -> x = 2;
constraint X1 = 5 -> x = 0;
solve :: seq_search([
  int_search([X1], input_order, indomain_min, complete)
]) satisfy;
output [show(x)];
```

Definition 1. A ClpZinc goal is either

- a constraint,
- a MiniZinc search annotation,
- a call to a user-defined predicate,
- the conjunction $(A;B)$ or the disjunction $(A;B)$ of two goals.

A ClpZinc clause is an item of the form `p(t1,...,tn) :- goal.` where t_1 and t_n are terms and goal is a ClpZinc goal. The goal part can be omitted: “`p(t1,...,tn).`” is a shorthand for “`p(t1,...,tn) :- true.`”.

The search annotations of MiniZinc are accessible in goals in order to allow the composition of user-defined strategies with built-in ones. Terms are either logical variables (x, y, Max, \dots), numbers, or compound terms of the form $p(t_1, \dots, t_n)$ where t_1, \dots , and t_n are terms. Model variables are a special case of compound terms, either atomic (a, b, \dots) or array accessors ($x[I, J]$). Zinc arrays have been unified with Prolog-like lists to ease their enumeration in search strategies.

In $\text{CLP}(\mathcal{X} + \mathcal{H})$, arithmetic differs from Prolog. Indeed, in accordance with the theory of CLP and unlike most Prolog systems, arithmetic is supposed to be contained in \mathcal{X} and is distinguished from \mathcal{H} terms, *e.g.*, “ $1 + 1$ ” is undistinguishable from “ 2 ” and is not a \mathcal{H} term. In ClpZinc, the different forms of unification, equality, and evaluation predicates that are encountered in Prolog systems ($=$, $\#$, is , \dots) are thus all unified in a unique notion of equality, which is accessible either explicitly with the predicate $=$, or implicitly when predicate arguments in either \mathcal{X} or \mathcal{H} are unified.

Arithmetic expressions are also extended for accessing the indexicals of the model variables. For instance, the goal $m = \text{min}(x)$ assumes that x is a model variable and unifies m with the currently known lower-bound of x . We consider the indexicals `min`, `max`, `card` and `dom_nth` (for retrieving the n th value in a variable domain). Concretely, an intermediary variable is introduced to receive the value of the indexical and search annotations are emitted for getting them with:

```
annotation indexical_min(var int: target, var int: x);
annotation indexical_max(var int: target, var int: x);
annotation indexical_card(var int: target, var int: x);
annotation indexical_dom_nth(var int: target, var int: x, var int: n);
```

These annotations require to extend the solvers to communicate the indexicals. That is the only change made to the interface of the solvers.

Example 2 (Dichotomic search). The Zinc `indomain_split` value selection strategy can be implemented in ClpZinc using indexicals. The predicate `dichotomy/3` below expresses the bisection of a variable x that has the initial domain `Min...Max`. The bisection defined in the auxiliary predicate `dichotomy/2` is iterated $\text{Depth} = \lceil \log_2 |x| \rceil$ times to ensure that the domain is reduced to a value on every leaf.

```
dichotomy(X, Min, Max) :-
    dichotomy(X, ceil(log(2, Max - Min + 1))).

dichotomy(X, Depth) :-
    Depth > 0,
    Middle = (min(X) + max(X)) div 2,
    (X <= Middle ; X > Middle),
    dichotomy(X, Depth - 1).
dichotomy(X, 0).

var 0..5: x;
:- dichotomy(x, 0, 5).
```

The MiniZinc model generated for the given goal is

```
var 0..5: x;
var 0..5: X3; var 0..5: X5; var 0..1: X7;
var 0..5: X4; var 0..5: X6; var 0..5: X2;
var 0..1: X8; var 0..5: X1; var 0..1: X9;
constraint X7 = 0 <-> x <= (X1 + X2) div 2;
constraint X8 = 0 <-> x <= (X3 + X4) div 2;
constraint X9 = 0 <-> x <= (X5 + X6) div 2;
```

```

solve :: seq_search([
  indexical_min(X1, x),
  indexical_max(X2, x),
  int_search([X7], input_order, indomain_min, complete),
  indexical_min(X3, x),
  indexical_max(X4, x),
  int_search([X8], input_order, indomain_min, complete),
  indexical_min(X5, x),
  indexical_max(X6, x),
  int_search([X9], input_order, indomain_min, complete)
]) satisfy;

```

The next example shows a partial search strategy that is not available using the usual MiniZinc search annotations. This is the interval slitting strategy introduced in [16] for solving Korf’s packing problem [10] (explained in Section 4) by making a preliminary coarse-grained filtering of the variable domains.

Example 3 (Interval splitting). The predicate `interval_splitting/4` defined below expresses the splitting of the domain of `x` into intervals of width `Step`. `x` is supposed to have the initial domain `Min...Max`.

```

interval_splitting(X, Step, Min, Max) :-
  Min + Step <= Max, NextX = min(X) + Step,
  (
    X < NextX
  ;
    X >= NextX,
    interval_splitting(X, Step, Min + Step, Max)
  ).
interval_splitting(X, Step, Min, Max) :-
  Min + Step > Max.
var 0..5: x;
:- interval_splitting(x, 2, 0, 5).

```

The corresponding MiniZinc model for the given goal is

```

var 0..5: x;
var 0..1: X3;
var 0..1: X4;
var 0..5: X2;
var 0..5: X1;
constraint X3 = 1 <-> x >= X1 + 2;
constraint X3 = 1 -> (X4 = 1 <-> x >= X2 + 2);
constraint X3 != 1 -> X4 = 0;
solve :: seq_search([
  indexical_min(X1, x),
  int_search([X3], input_order, indomain_min, complete),
  indexical_min(X2, x),
  int_search([X4], input_order, indomain_min, complete)
]) satisfy;

```

2.2 Partial Evaluation of ClpZinc into And/Or Trees

From now on, let us assume that the initial ClpZinc goals provided in the items “:- *goal*.” of the ClpZinc models that we consider, always terminate. That hypothesis should hold even if \mathcal{X} only resolves fully instantiated constraints, as is the case of the static partial evaluator. Verifying termination of logic programs is a classical topic for which many results have been obtained using type systems or abstract interpretation techniques [3]. The description of these techniques is however beyond the scope of this paper.

Given a constraint system \mathcal{X} , the partial evaluation of a $\text{CLP}(\mathcal{X} + \mathcal{H})$ goal will lead to an and/or tree with constraints over \mathcal{X} . The partial evaluator resolves predicate calls, Herbrand constraints and fully instantiated arithmetic constraints, *i.e.*, arithmetic tests. Since, without loss of generality, we settled for a DFS left-to-right CLP evaluation, the and/or trees will be traversed in a similar DFS left-to-right fashion in our examples, but any other traversal order can be treated similarly.

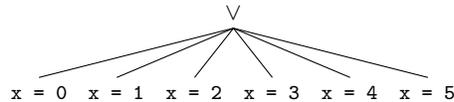


Fig. 1. And/Or tree derived from Example 1 (Labeling)

As shown in Figure 1 for Example 1, or-nodes are flattened so that nested choices become a single large disjunction. And-nodes are similarly flattened into conjunctions. In the general case, the partial evaluation of the continuation may duplicate constraints with different partial instantiations. For instance, Figure 2 shows a simple example of duplication with partial instantiation of the bounding constraint $\text{Min} \leq x, x \leq \text{Max}$.

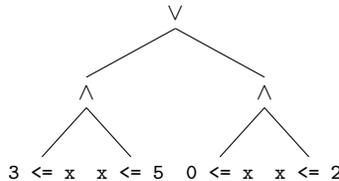


Fig. 2. And/Or tree for the ClpZinc goal

```
var 0..5: x;
:- (Min=3, Max=5; Min=0, Max=2), Min <= x, x <= Max.
```

However, when the partial evaluation store is left unchanged by a choice (typically, when only constraints in \mathcal{X} are involved), the continuation will remain undeveloped, as shown in Figure 3 for Example 2. The and/or tree is in logarithmic size with respect to the size of the domain whereas the fully expanded search tree would be in linear size.

We assume that the partial evaluation terminates and that the resulting and/or tree meets the following conditions:

1. all the variables that appear in constraints are finite-domain variables;

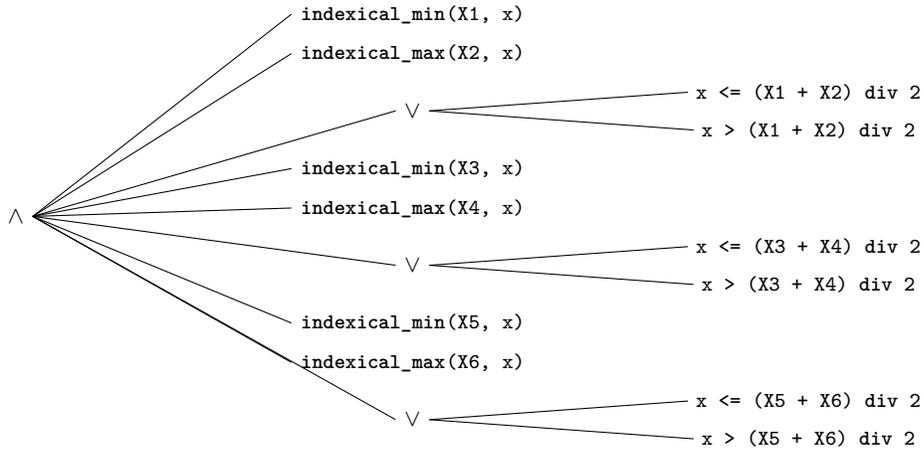


Fig. 3. And/Or tree for Example 2 (Dichotomic search)

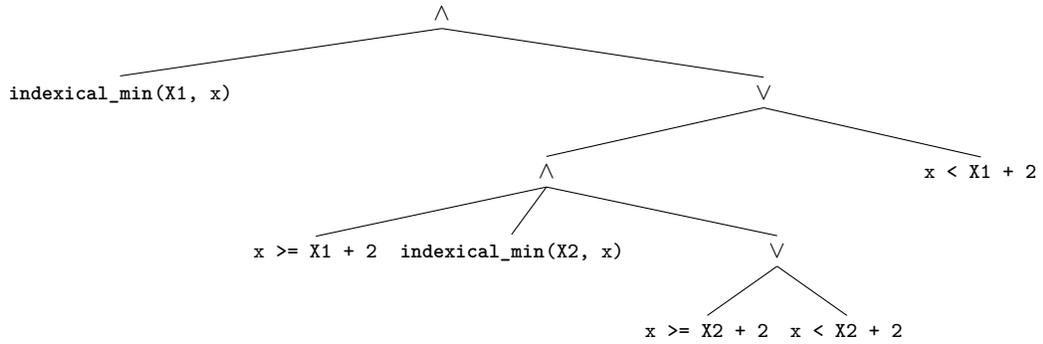


Fig. 4. And/Or tree for Example 3 (Interval splitting)

2. all lists are well-formed, in particular the tail of every non-empty list is a list and cannot be a variable since such a variable would be a finite-domain variable according to the previous condition (this ensures that lists can be expanded in Zinc array literals);
3. all annotations except indexicals do not appear below a choice point (*i.e.* their execution is unconditional).

3 Compiling And/Or Trees into Zinc Reified Constraints

Given a CSP(\mathcal{X}) model \mathcal{M} and a tree search strategy represented by the traversal of an and/or tree t , the generation of Zinc code proceeds by assigning an additional

model variable to every or-node in the tree t , and by emitting search annotations that fix the enumeration strategy for these additional variables in a way compatible with the traversal ordering.

In Figure 1, the variable x_1 is assigned to the root node, with the domain $0..5$ corresponding to the arity of the node. As shown in the Zinc model generated for Example 1, each constraint labeling the leaves under this or-node appears in the model guarded by an implication checking for a particular value of x_1 . Therefore, when the search annotation `int_search` enumerates the possible values of x_1 , these guarded constraints are successively enabled for exploring the different branches of the tree.

More generally, the transformation presented in this paper can be seen as a constructive proof for the following theorem. We call *fixed enumeration strategies* the search strategies that are reduced to a sequence of variables selected in a fixed order and enumerated with the increasing value selection (`indomain_min`). For dynamic search strategies, this sequence is possibly interleaved with accesses to indexicals.

Theorem 1. *For every pair (\mathcal{M}, t) where \mathcal{M} is a CSP model and t a tree search strategy described as the traversal of an and/or tree, there exists a model \mathcal{M}_t and a fixed enumeration strategy t' such that the resolution of $(\mathcal{M} + \mathcal{M}_t, t')$ explores the same search tree as (\mathcal{M}, t) .*

Proof. First, let us remark that in $\mathcal{M} + \mathcal{M}_t$, the variables and the constraints of \mathcal{M} are left unchanged; only additional model variables accompanied with additional constraints are introduced in \mathcal{M}_t .

Let us assume that we have a function ℓ that maps each or-node n of t to a model variable $\ell(n) \in V(\mathcal{M}_t)$, such that for every pair n_1, n_2 of nodes of t , if $\ell(n_1) = \ell(n_2)$, either $n_1 = n_2$ or the lowest common ancestor of n_1 and n_2 is an or-node.

Each constraint c that appears as a leaf of t is translated as a constraint in \mathcal{M}_t . Let n_1, \dots, n_k denote the or-nodes that are traversed by the path π from the root of t to the leaf c and, for every $1 \leq i \leq k$, let p_i be the rank of the branch taken by π at node n_i . We adopt the convention that branches are numbered from left to right and that the left-most branch has rank 0. Then the following constraint is posted in the MiniZinc model, for translating the leaf c :

$$\text{constraint } \ell(n_1)=p_1 \wedge \dots \wedge \ell(n_k)=p_k \rightarrow c;$$

Let $X \in V(\mathcal{M}_t)$ be one of the variables that label or-nodes. The domain of X will be $0.. \max\{w(n) - 1 \mid \ell(n) = X\}$ where $w(n)$ denotes the width of the or-node n (i.e., the number of branches issued from n). For every or-node n_k such that $\ell(n_k) = X$ that does not reach this maximum, the following additional constraint is posted, where $(n_i, p_i)_i$ denotes the or-path to n_k as above:

$$\text{constraint } \ell(n_1)=p_1 \wedge \dots \wedge \ell(n_{k-1})=p_{k-1} \rightarrow \ell(n_k) < w(k);$$

We should now establish the connection between the enumeration of the variables that label the or-nodes and the exploration of the and/or tree.

Search annotations have to be emitted to fix the enumeration strategy for the variables $V(\mathcal{M}_t)$. MiniZinc search annotations have a depth-first semantics. To reproduce the semantics of (\mathcal{M}, t) , it is thus sufficient in t' to emit the annotations that select the variables in the order where their corresponding nodes are encountered following the traversal of t . The value selection strategy fixes the order in which the sub-branches are explored. For t' , it can thus be reduced to the value selection `indomain_min` that selects the left-most branch, by switching the values if necessary. We thus have that, by construction, $(\mathcal{M} + \mathcal{M}_t, t')$ explores the same search tree as (\mathcal{M}, t) . \square

The two following optimizations are not mandatory but have been measured to give significant performance improvements:

1. to prevent enumerating on X in branches where X does not occur, the following constraint imposes a fixed value to X on these branches.

$$\text{constraint } \left(\bigwedge_{\substack{(n_i, p_i)_i \\ \ell(n_k)=X}} \ell(n_1) \neq p_1 \vee \dots \vee \ell(n_k) \neq p_k \right) \rightarrow X=0;$$

2. in the particular case where a constraint c occurs under an or-node n_k (possibly separated with some and-nodes) and when $\neg c$ occurs in every other branches of n_k , then the following constraint is posted instead (and the constraints corresponding to the leaves $\neg c$ are not posted).

$$\text{constraint } \ell(n_1)=p_1 \wedge \dots \wedge \ell(n_{k-1})=p_{k-1} \rightarrow (\ell(n_k)=p_k \leftrightarrow c);$$

These simplifications can be seen in the Zinc code generated for Examples 2 and 3.

4 Computation Results on Korf's Square Packing Benchmark

In this section, we consider Korf's Optimal Rectangle Square Packing problem [10], *i.e.*, given an integer $n \geq 1$, find an enclosing rectangle of smallest area containing n squares from sizes 1×1 , 2×2 , up to $n \times n$, without overlap. Helmut Simonis and Barry O'Sullivan proposed a complex dynamic search strategy for that problem in [16], which is interesting to specify and evaluate in ClpZinc.

First, the model they consider for packing the n consecutive squares in a rectangle of size $w \times h$ can be written in MiniZinc as follows. Since the 1×1 square can always be placed afterward if the area $w \times h$ is big enough, the model only considers the remaining $n - 1$ other squares. Two redundant `cumulative` constraints are introduced. The two last constraints break some symmetries by forcing the largest square to be in the bottom-left quadrant.

```
int: n;
constraint diffn(x,y,[i+1|i in 1..n-1],[i+1|i in 1..n-1]);
```

```

constraint cumulative(x,[i+1|i in 1..n-1],[i+1|i in 1..n-1],h);
constraint cumulative(y,[i+1|i in 1..n-1],[i+1|i in 1..n-1],w);

constraint forall(i in 1..n-1) (x[i] <= w - i /\ y[i] <= h - i);
constraint x[n-1] <= (w - n + 2) div 2;
constraint y[n-1] <= (h + 1) div 2;

```

Second, the optimization procedure used in [16] enumerates all the possible sizes $w \times h$ for the enclosing rectangle by increasing area. This strategy can be internalized in the model by successively considering all the rectangles up to $\text{max_size} \times \text{max_size}$, from the minimal area covered by the squares themselves ($\sum_{1 \leq i \leq n} i^2$) and with bounds on w and h that are described in [16]:

```

int: max_size;
array[1..n-1] of var 1..max_size: x;
array[1..n-1] of var 1..max_size: y;
var 0..max_size: w; var 0..max_size: h;
var 0..max_size * max_size: area;

constraint w * h = area /\ w <= h;
constraint sum([i*i | i in 1..n]) <= area;
constraint w >= 2 * n - 1
  /\ h >= (n * n + n - ((w + 1) div 2 - 1) * ((w + 1) div 2 - 1)
    - ((w + 1) div 2 - 1)) div 2;

```

Now, the search strategy of [16] firsts enumerates in x and then in y , considering in each dimension a preliminary interval splitting on the origins of the squares from sizes $n \times n$ to 7×7 , and then a dichotomic search on the origins, still by considering the biggest square first. This search strategy is implemented in ClpZinc by enumerating first on area and w to find the rectangle of smallest area first. It is worth noticing that we can combine the user-defined interval splitting strategy defined in Example 3 with the built-in dichotomic search (`indomain_split`).

```

interval_splitting_list(L, S, Stop) :-
  (S <= Stop ; S > Stop, L = []).
interval_splitting_list([H | T], S, Stop) :-
  S > Stop,
  interval_splitting(H, max(1, (S * 3) div 10) + 1, 0, max_size),
  interval_splitting_list(T, S - 1, Stop).

:- int_search([area, w], input_order, indomain_min, complete),
   reverse(x, RXs), interval_splitting_list(RXs, n, 6),
   int_search(RXs, input_order, indomain_split, complete),
   reverse(y, RYs), interval_splitting_list(RYs, n, 0),
   int_search(RYs, input_order, indomain_split, complete).

```

This strategy can be compared to the use of the dichotomic search only, on each dimension, from the biggest to the smallest square, relying on the native `indomain_split` of MiniZinc. This is indeed a good candidate for the best strategy that can be easily written in MiniZinc without the help of ClpZinc.

```

solve :: seq_search([
  int_search([area, w], input_order, indomain_min, complete),
  int_search([x[n-1] | i in 1..n-1] ++ [y[n-1] | i in 1..n-1],
    input_order, indomain_split, complete)
]) satisfy;

```

To measure the overhead of ClpZinc, we also include the version of dichotomic search relying on the user-defined predicate of Example 2.

```

:- int_search([area, w], input_order, indomain_min, complete),
   reverse(x, Rx), dichotomy_list(Rx, 0, max_size),
   reverse(y, Ry), dichotomy_list(Ry, 0, max_size).

```

n	Choco 3			SICStus	
	dichotomic indomain_split	dichotomic ClpZinc	interval split then dichotomic ClpZinc	interval split then dichotomic ClpZinc	interval split then dichotomic Original
16	9 232	14 402	853	710	340
17	16 321	21 643	982	450	250
18	422 116	570 407	7 978	9 400	4 850
19	785 080	1 051 418	6 984	11 710	4 310
20			12 572	17 330	8 970
21			42 892	88 310	32 370
22			208 632	303 810	153 860
23			1 340 816	2 104 020	999 020
24			2 312 933	3 433 410	1 481 910
25			29 201 522	38 310 220	10 662 860
26			142 702 128	195 378 230	62 179 600

Table 1. Solving times in ms for Korf’s problem for strategies implemented in ClpZinc with Choco 3 and SICStus as solvers, compared to the original SICStus program.

Table 1 shows the results of the native dichotomic search procedure in MiniZinc, of the user-defined dichotomic and interval-splitting search procedure in ClpZinc, solved either Choco or SICStus solvers, and of the original SICStus-Prolog program of [16], all of them running on Intel®Xeon®CPU E5-1620 0 @ 3.60GHz machines. As shown in Table 1, the overhead introduced by the reification of the search procedure is quite reasonable, averaging a two-fold slowdown of the program. On the other hand, the reified search enables the encoding of the interval splitting strategy that induces a crucial increase in performance comparable to the results obtained in [16].

That table also shows that the specification in ClpZinc of the dichotomic and interval-splitting search strategy makes it readily available in a variety of solvers for which its implementation was not trivial. The implementation in Choco is the most efficient, followed by SICStus-Prolog, probably due to differences in the implementation of reified constraints.

5 LDS and SBDS as Strategy Transformers in ClpZinc

Since and/or trees are first-class terms in ClpZinc, they can be arguments of ClpZinc predicates to define search strategy transformers. In this section, we illustrate this possibility with the modeling of Limited Discrepancy Search (LDS) [9] and Symmetry Breaking During Search (SBDS) [7] as strategy transformers

for labeling or dichotomic search for instance. This technique is closely related to the monadic approach of strategy transformers presented in [15]. The main difference, outside of purely syntactic choices, is that the monadic transformations described in [15] heavily rely on laziness to not expand the trees, whereas in ClpZinc, in order to finally compile towards a CSP, we fully meta-interpret, and therefore expand, the search trees, with some possible benefits thanks to the propagation of search constraints.

5.1 Limited Discrepancy Search

LDS can be modeled very simply in ClpZinc using meta-interpretation. Basically the and/or tree is developed but the right turns are counted at the same time, by increment when going in the right branch of an *or* and by addition of the two branches when going through an *and*:

```
lds(true, L).
lds((A ; B), L) :-
    domain(L0, 0, 1024), domain(D, 0, 1),
    ( D = 0, lds(A, L0)
    ; D = 1, lds(B, L0)),
    L = D + L0.
lds((A, B), L) :-
    domain(L0, 0, 1024), domain(L1, 0, 1024),
    lds(A, L0), lds(B, L1),
    L = L0 + L1.
lds(B, L) :- builtin(B), B, L = 0.
lds(H, L) :- clause(H, B), lds(B, L).
```

Interestingly, since right turns are counted at the constraint level, the propagation of search constraints may actively reduce the search space, whereas a classical procedural implementation of LDS limits the number of right turns by generate-and-test. The following example demonstrates an *exponential speed-up* thanks to this propagation with respect to a procedural implementation of LDS.

```
var 0..1: x;
var 0..1: y;
array[0..n] of var 0..1: a;

:- int_search(a, input_order, indomain_min, complete),
   lds(((x = 0; x = 1), (y = 0; y = 1)), 0), x != y.
```

Whereas a procedural implementation would explore the 2^n possible assignments for *a* before detecting that the model is unsatisfiable within the reduced search space, the inconsistency is immediately detected in the MiniZinc model generated by ClpZinc.

```
var 0..1: x;
var 0..1: y;
array[0..n] of var 0..1: a;
var 0..1024: X3;
var 0..1024: X5;
var 0..1: X7;
var 0..1: X4;
var 0..1: X6;
var 0..1024: X2;
var 0..1: X8;
var 0..1024: X1;
constraint X7 = 0 -> X4 = 0;
constraint X7 = 0 -> x = 0;
constraint X7 = 0 -> X3 = 0;
```

```

constraint X7 = 1 -> X4 = 1;
constraint X7 = 1 -> x = 1;
constraint X7 = 1 -> X3 = 0;
constraint X1 = X4 + X3;
constraint X8 = 0 -> X6 = 0;
constraint X8 = 0 -> y = 0;
constraint X8 = 0 -> X5 = 0;
constraint X8 = 1 -> X6 = 1;
constraint X8 = 1 -> y = 1;
constraint X8 = 1 -> X5 = 0;
constraint X2 = X6 + X5;
constraint 0 = X1 + X2;
constraint x != y;
solve :: seq_search([
  int_search(a, input_order, indomain_min, complete),
  int_search([X7], input_order, indomain_min, complete),
  int_search([X8], input_order, indomain_min, complete)
]) satisfy;
n = 1000;

```

5.2 Symmetry Breaking During Search

Symmetry Breaking During Search [2,7] is a general method that transforms a search tree so as to remove symmetric branches from enumeration. Each time the search backtracks from enumerating solutions with given a search constraint c , the other search branch considers $\neg c$ and also all the symmetric constraints $\sigma(\neg c)$ for symmetries σ compatible with search constraints already posted. This schema is implemented in the predicate below, supposing a predicate `cut_symmetry` that adds the symmetric negations for a given constraint.

```

sbds(top, _).
sbds(or(A, B), Path) :-
  ( A = constraint(C, A0),
    ( C, sbds(A, [C | Path])
      ; cut_symmetry(C, Path), sbds(B, Path))
    ; A \= constraint(_, _),
      (sbds(A, Path) ; sbds(B, Path))).
sbds(constraint(C, T), Path) :- C, sbds(T, [C | Path]).
:- search_tree(labeling_list(queens, 1, n), T), sbds(T, []).

```

The predicate `search_tree` constructs the search tree associated with the and/or tree of a CLP goal by meta-interpretation (full code in Appendix A).

6 Beyond Tree Search Strategies

Some search strategies require to iterate a search tree several times with a memory passed from one branch to another. That is typically the case for optimization methods like branch-and-bound where the best score reached up to now is remembered from one iteration to another of the underlying search strategy, or for shaving, where one step of propagation is performed and undone in order to select the best one. In languages like Prolog, such methods are implemented with the help of a global state, most commonly stored within the fact database (with `assert` and `retract`). We propose two additional annotations for search in MiniZinc to handle global state.

```

annotation store(var bool: c, string: id, array[int] of var int: src);
annotation retrieve(string: id, array[int] of var int: target);

```

The semantics of `store(cond, id, source)` is to remember, if `cond` is `true`, the current values of the sequence of variables `source` into the global state identified as `id`. The `store` annotation does nothing if `cond` is false, such that the assignment to `id` is skipped outside the computation branch that involves this assignment. The parameter `cond` does not appear in ClpZinc: it is implicitly fixed to the guard associated to the path leading to the node where the annotation appears in the and/or tree. The semantics of `retrieve(id, target)` is to assign the values previously remembered into the global state identified as `id` into the sequence of variables `target`.

As shown below, these two simple annotations allow the specification of branch-and-bound optimization in ClpZinc. Once again, for such strategy one might also use the native `maximize` annotation of MiniZinc, but as far as we know, more complex iterative procedures like shaving or enumerating solutions, using previously found ones in the search (whether to guide it or to limit it), cannot be natively written in MiniZinc.

```

maximize(G, S, Min, Max) :-
  domain(I, Min, Max + 1), domain(Best, Min, Max),
  domain(Fail, 0, 1),
  domain(A, 0, 1), domain(B, 0, 1), domain(C, 0, 1),
  (Fail = 0 -> A != B /\ B != C /\ A != C),
  store("bb_best", [Min, 0]),
  labeling(I, Min, Max + 1),
  retrieve("bb_best", [Best, Fail]),
  ( Fail = 0, store("bb_best", [Best, 1]),
    S > Best, G, store("bb_best", [S, 0]),
    labeling(A, 0, 1), labeling(B, 0, 1)
  ; Fail = 1, I = Max + 1, S = Best, G).

minimize(G, S, Min, Max) :-
  domain(Dual, Min, Max), Dual = Max - S + Min,
  maximize(G, Dual, Min, Max).

```

Note that in order to make this branch-and-bound procedure possible, the gap between failures at the search and at the constraint level has to be bridged. Using the incompleteness of arc-consistency, the reified constraint imposing that `A`, `B` and `C` are all different allows us to fail at will in the success branches (`Fail = 0`) by labelling `A` and `B`. There is also an optimization in the above code where the upper bound on the score is used in the `Fail = 1` branch as some kind of *cut*: all attempts after the first failure will be immediately discarded, except the last one where appropriate values for variables will be rebuilt by running the goal `G` again.

7 Conclusion

We have shown that tree search procedures, such as for instance heuristic labeling, dichotomy, interval-splitting, limited discrepancy search, and dynamic symmetry breaking during search, can be internalized in a constraint model through reified constraints. On the complex dynamic strategy used for solving Korf's benchmark for square packing, we have shown that the implementation overhead is limited to a factor 2, and can be measured on different CSP solvers without particular support for search. We have also shown with an example that the propagation of

search constraints can in fact exhibit an exponential speed up, compared to a classical procedural implementation of the search strategy.

This has been demonstrated by realizing an extension of MiniZinc with CLP clauses. Annotations for indexicals have also been added to MiniZinc for defining dynamic search strategies. Furthermore, by adding annotations for storing intermediate values during search, we have shown that this approach can be generalized to non tree search procedures, such as branch-and-bound optimization.

It is worth noting that the conversion of search into constraints opens up a whole field of challenges for CSP solvers with limited built-in search strategies. For instance, Korf’s packing problem with the complex strategy of [16] can now be proposed for the MiniZinc contest, since any constraint solver implementing the indexical *min* can in principle solve it. Therefore, the two sentences of [16] stating that this packing problem “nicely tests the generality of a search method” and is a “more attractive benchmark for placement problems than the perfect square” can now apply to compare a broad range of CSP solvers. Furthermore, work on complex problems with dedicated heuristics can now become more independent of a particular solver through the modeling of the search strategy in our approach. We have added the needed indexicals to the FlatZinc parser of some solvers (Choco [5], JaCoP [11], SICStus [1], Gecode [4], or-tools [8]) and encourage all solver developers to do so in order to tackle these new challenging problems for the MiniZinc community.

Finally, as a perspective for future work, the reification of choice point constraints in our scheme is in principle compatible with lazy clause generation techniques [13] and the learning of nogood by using a SAT solver. Such a combination of modeling search by constraints and learning constraints during search is however quite intriguing and will be the matter of future work.

Acknowledgments

We are grateful to the French ANR for its support of the Net-WMS-2 project, and to the partners of this project for stimulating discussions on these topics.

References

1. SICS AB. Sicstus 4.2.3, 2012.
2. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999.
3. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.
4. The Gecode Community. Gecode 4.2.1, 2013.
5. École des Mines de Nantes. Choco 3, 2014.
6. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

7. Ian P. Gent and Barbara Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, pages 599–603, 1999.
8. Google. or-tools, 2014.
9. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence*, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
10. Richard E. Korf. Optimal rectangle packing: Initial results. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003.
11. Krzysztof Kuchcinski and Radoslaw Szymanek. Jacop 4.0.0, 2013.
12. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.
13. O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 16(9):357–391, 2009.
14. Reza Rafeh, Kim Marriott, Maria Garcia de la Banda, Nicholas Nethercote, and Mark Wallace. Adding search to zinc. In *CP*, pages 624–629, 2008.
15. Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663, 2009.
16. Helmut Simonis and Barry O’Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *Proceedings of CP'08*, volume 5202 of *LNCS*, pages 52–66. Springer-Verlag, 2008.
17. Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.

A ClpZinc Model of Symmetry Breaking During Search

The full code for SBDS on the N-queens example, including the transformation of the and/or tree to a search tree during meta-interpretation.

```

int: n;
array[1..n] of var 1..n: queens;

include "globals.mzn";
include "prolog.plz";

constraint all_different(queens);
constraint all_different([queens[i] + i | i in 1..n]);
constraint all_different([queens[i] - i | i in 1..n]);

labeling(X, Min, Max) :-
    Min <= Max,
    (
        X = Min
        ;
        labeling(X, Min + 1, Max)
    ).

labeling_list([], Min, Max).
labeling_list([H | T], Min, Max) :-
    labeling(H, Min, Max),
    labeling_list(T, Min, Max).

builtin(true).
builtin(false).
builtin(_ = _).

```

```

builtin(_ <= _).
builtin(_ > _).

sequence_tree(top, B, T) :-
    search_tree(B, T).

sequence_tree(bot, _T, bot).

sequence_tree(or(A0, B0), T, or(A1, B1)) :-
    sequence_tree(A0, T, A1),
    sequence_tree(B0, T, B1).

sequence_tree(constraint(C, T0), T, constraint(C, T1)) :-
    sequence_tree(T0, T, T1).

search_tree((A, B), T) :-
    search_tree(A, TA),
    sequence_tree(TA, B, T).

search_tree((A ; B), T) :-
    search_tree(A, TA),
    search_tree(B, TB),
    (
        TA == false,
        TB == false,
        T = bot
    ;
        TA == false,
        TB \= false,
        T = TB
    ;
        TA \= false,
        TB == false,
        T = TA
    ;
        TA \= false,
        TB \= false,
        T = or(TA, TB)
    ).

search_tree(B, T) :-
    builtin(B),
    (
        B == true,
        T = top
    ;
        B == false,
        T = bot
    ;
        B \= true,
        B \= false,
        T = constraint(B, true)
    ).

search_tree(H, T) :-
    findall(B, clause(H, B), L),
    L \= [],
    disjunction(L, D),
    search_tree(D, T).

disjunction([], false).
disjunction([X], X).
disjunction([A, B | T], (A; C)) :-
    disjunction([B | T], C).

```

```

sbds(top, _).
sbds(or(A, B), Path) :-
  (
    A = constraint(C, A0),
    (
      C,
      sbds(A, [C | Path])
    ;
      cut_symmetry(C, Path),
      sbds(B, Path)
    )
  )
;
A \= constraint(_, _),
(
  sbds(A, Path)
;
  sbds(B, Path)
)
).

sbds(constraint(C, T), Path) :-
  C,
  sbds(T, [C | Path]).

:- search_tree(labeling_list(queens, 1, n), T), sbds(T, []).

cut_symmetry(C, Path) :-
  (
    C = (queens[I] = J),
    (
      Path = [],
      queens[I] != n - J + 1
    ;
      Path \= []
    )
  )
;
C \= (queens[I] = J),
true
).

output [
  show(queens) ++ "\n"
] ++ [
  if j = 1 then "\n" else "" endif ++
  if fix(queens[i]) = j then
    show_int(2,j)
  else
    "--"
  endif
  | i in 1..n, j in 1..n
] ++ ["\n"];

n = 3;

```