Search by Constraint Propagation

Thierry Martinez François Fages Sylvain Soliman

Inria Paris-Rocquencourt, France

Thierry.Martinez@inria.fr Francois.Fages@inria.fr Sylvain.Soliman@inria.fr

Abstract

Constraint programming is traditionally presented 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 a Horn clause extension of the MiniZinc modeling language and the modeling in this language 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.

Keywords modeling languages, search, constraint programming, Horn clauses

1. Introduction

Constraint programming (CP) is traditionally presented as the combination of two components: a constraint model and a search procedure [22]. The search procedure is often crucial to solve hard combinatorial problems in combination with constraint propagation. However its definition is hardly declarative and most CP systems rely on using imperative programs in the host programming language for this part. There has been attempts to make search declarative with the definition of search combinators [20], list monads in the functional settings [3], or meta-interpreters in logic programming [13], but these approaches remain limited in expressive power or performance.

On the other hand, declarative front-end modeling languages have been designed to express the constraint modeling part easily and independently of the solvers used to solve them. For using a constraint programming solver, such modeling languages have thus to rely on either a fixed search strategy, e.g. Essence [9], or special annotations for specifying the search strategy, e.g. MiniZinc [16, 18]. This modeling language MiniZinc [1] succeeded in being widely used in the constraint programming community and in becoming a common input format for using and comparing many solvers, as well as for defining constraint satisfaction problem (CSP) contests. In MiniZinc, the annotations for the search procedure are dedicated to the constraint programming solvers and are ignored by the other solvers.

In this paper, we show that a completely different approach for specifying search in constraint modeling languages is possible, by *internalizing the search strategy in the constraint model* and using a standard labeling 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;
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 Boolean 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. We show this approach is not limited to static search procedures in which all choice points are precisely known in advance, but can accommodate dynamic search strategies, such as dichotomic or interval splitting search [21] for example. In constraint programming, dynamic search procedures rely on the values of indexicals (domain size, minimum value, etc.). In the framework presented here, they are expressed by extending the enumeration strategy with annotations for assigning the values of indexicals to auxiliary model variables. Static search procedures do not rely on the values of indexicals and their encodings 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.

To make concrete the presentation of the transformation, we consider the MiniZinc modeling language and introduce ClpZinc [2], a language extension of MiniZinc with the ability to describe new relations by Horn clauses, similarly to Constraint Logic Programs (CLP). The choice of this language of Horn clauses is motivated by the need for specifying search procedures to express constraint

PPDP '15, July 14–16, 2015, Siena, Italy.

http://dx.doi.org/10.1145/2790449.2790527

[1] `http://www.minizinc.org/`

[2] The Clp2Zinc compiler transforms ClpZinc models in MiniZinc models. It is available for download, together with patches for the solvers Choco, JaCoP, SICStus, Gecode and or-tools: `http://lifeware.inria.fr/~tmartine/clp2zinc/`

predicates (for tests and constraint posting), conjunction, disjunction (for tests and search points), and recursion (for iteration). Given a constraint system $\mathcal{X}$ (e.g. finite domains) and the Herbrand constraint system $\mathcal{H}$, we consider the language of CLP($\mathcal{X}+\mathcal{H}$) Horn clauses and describe a partial evaluation procedure to transform any terminating CLP($\mathcal{X}+\mathcal{H}$) goal to an and/or tree with constraints over $\mathcal{X}$. From the point of view of CLP, this is an alternative execution model which proceeds by goal expansion with constraint solving over $\mathcal{H}$ only, instead of by the standard CSLD($\mathcal{X}+\mathcal{H}$) resolution procedure [13].

The paper is organized as follows. In Section 2, we first introduce the encoding of the dichotomic search procedure into constraints, showing that a couple of arithmetic constraints suffice with a number of variables logarithmic in the size of the domain. In Section 3, we generalize this approach by encoding any search strategy described as an and/or tree into reified constraints. In Section 4, we introduce the ClpZinc language for describing search strategies and describe the transformation of search procedures from CLP($\mathcal{X}+\mathcal{H}$) to and/or trees over $\mathcal{X}$. In the subsequent sections, we evaluate this approach on benchmarks of models with specific search strategies, namely: Korf's Square Packing problem in Section 5, limited discrepancy search in Section 6.1 and symmetry breaking during search in Section 6.2. In Section 7, 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 8, we compare our approach with Search Combinators, which are another proposal for describing search strategies in MiniZinc. Finally, we conclude and give some new perspectives for future work.

2. Dichotomic Search and Interval Splitting with two Arithmetic Constraints

Let x be an integer variable in the domain $[0, 2^d[$. Dichotomic search in this domain amounts to explore a binary search tree of height d in which each node is associated with a restricted interval domain of size divided by two at each level, up to single values in the leaves. Dichotomic search is thus equivalent to the enumeration of the values of the bits in the writing of x in base 2, from the most significant bit to the less significant one. This remark leads to an implementation of dichotomic search which consists in posting the equality constraint

$$x = \sum_{0 \leq k < d} x_k 2^k$$

where the auxiliary variables x_k are Boolean in $\{0, 1\}$, and simply enumerating the values of those variables taken in the order x_{d-1}, x_{d-2}, up to x_0. It is worth noticing that the number of auxiliary Boolean variables in this implementation is only logarithmic in the size of the domain. The dichotomic search tree obtained by constraint propagation and domain filtering is depicted in Figure 1.

Interval splitting is a complementary search strategy for making a coarse grain interval assignment before proceeding by dichotomic search. It is used for instance in the packing problems presented in Section 5 for performance evaluation, to make first a coarse-grained placement of objects by splitting each dimension into intervals up to some length, and then try to refine it by dichotomic search up to a precise placement. Let x be an integer variable in some domain $[0, n[$ and s be the size of the intervals into which the domain should be split. Interval splitting on x can be implemented as an Euclidean division with denominator s by first posting the equality constraint

$$x = s \times q + r$$

where the rest r is an auxiliary integer variable in $[0,\ s[$, and the quotient q in $[0, \lceil n/s \rceil [$ gives the intervals of size s where x lays in, and second by simply enumerating the values of q. The interval split tree is depicted in Figure 2.

We show in the next section that these transformations of simple search strategies into arithmetic constraints with standard labeling strategies generalize to tree search procedures.

3. Compiling And/Or Trees into Reified Constraints

Many search procedures correspond to the traversal of an ordered and/or tree where the leaves are labeled by constraints over some domain X. In this section we show that any such search procedure can be implemented with the standard labeling strategy in a model augmented with reified constraints.

Definition 1. *The set of ordered* and/or trees *over a constraint domain $\mathcal{X}$ is the least set*

- *containing constraints over $\mathcal{X}$, noted c,*
- *and closed by ordered conjunction, noted $t \wedge t'$ where t and t' are two and/or trees,*
- *and ordered disjunction, noted $t \vee t'$ where t and t' are two and/or trees.*

The standard *labeling* and/or tree is composed of a conjunction of disjunction of equality constraints for each variable, between that variable and the different values in its domain in increasing order (increasing value selection strategy `indomain_min`).

Definition 2. *The set of ordered* search trees *over a constraint domain $\mathcal{X}$ is the least set*

- *containing constraints over $\mathcal{X}$,*
- *and closed by labeled ordered node formation, noted $(c, t \vee t')$ where c is a constraint over $\mathcal{X}$ and t and t' are two search trees.*

For a constraint c and a search tree t, we denote by $c \wedge t$ the search tree $c \wedge c'$ if $t = c'$ is a constraint, and $(c \wedge c', t_0 \vee t_1)$ if $t = (c', t_0 \vee t_1)$. We denote by $\top$ the constraint true.

An and/or-tree can be transformed to a search tree by a standard transformation performed for instance by CSLD resolution in CLP [13]. For practical reasons, we define here this transformation by structural induction as a function from sequences of and/or-trees to search trees.

Definition 3. *The transformation function s from sequences of and/or-trees to search trees is defined by structural induction by*

- $s(\epsilon) = \top$,
- $s(c \cdot \kappa) = c \wedge s(\kappa)$
- $s((t \wedge t') \cdot \kappa) = s(t \cdot t' \cdot \kappa)$
- $s((t \vee t') \cdot \kappa) = (\top, s(t \cdot \kappa) \vee s(t' \cdot \kappa))$

Given a constraint model $\mathcal{M}$ over $\mathcal{X}$ and a tree search strategy represented by an and/or tree t, we associate a constraint model M' in which a Boolean variable and a reified constraint is added for each or-node in t and the standard labeling strategy is equivalent to t. That transformation is given here by the constructive proof of the following theorem in which the reified constraint are explicited.

Definition 4. *Given a constraint model $\mathcal{M}$ over $\mathcal{X}$, two search trees t and t' over $\mathcal{X}$ are* equivalent w.r.t. $\mathcal{M}$, *noted $t \equiv_{\mathcal{M}} t'$, if:*

- *t and t' have the same tree structure;*
- *for all path π from the root to a node in t, we have $\mathcal{M} \vDash c \Leftrightarrow c'$, where c is the conjunction of constraints labeling the nodes along π projected over the variables of $\mathcal{M}$ and c' is the projected conjunction of constraints for the same path π in t'.*

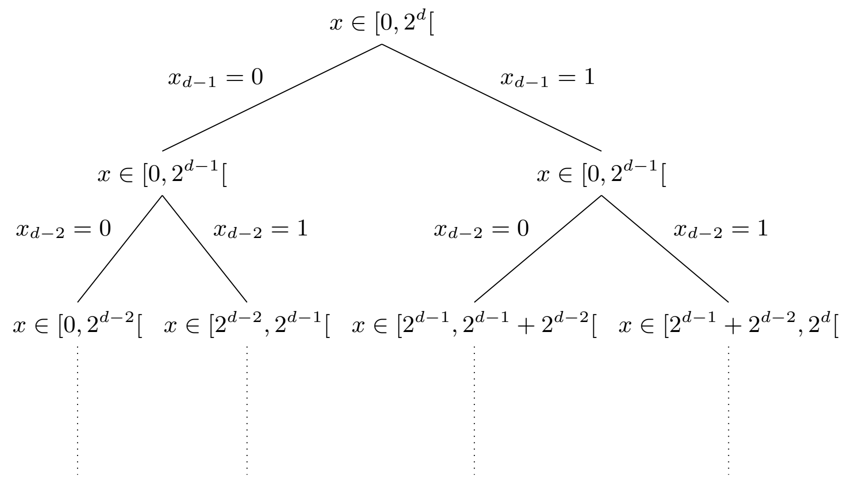


Figure 1. Dichomotic search tree obtained by domain filtering and constraint propagation of the equality $x = \sum_{0 \leq k < d} x_k 2^k$, with a simple labeling of the Boolean variables $x_{d-1}, x_{d-2}, \ldots, x_0$ in that order.

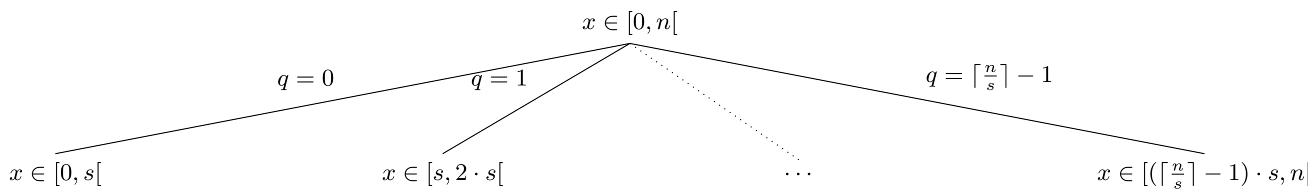


Figure 2. Interval splitting tree obtained by domain filtering and constraint propagation of the equality $x = s \times q + r$ where $r \in [0, s[$, with a simple labeling of the integer variable q in $[0, \lceil \frac{n}{s} \rceil[$.

Theorem 1. *For every pair $(\mathcal{M}, t)$ where $\mathcal{M}$ is a constraint model and t an and/or tree, there exists a model $\mathcal{M}_t$ and a labeling and/or tree t' such that $s(t) \equiv_{\mathcal{M} \cup \mathcal{M}_t} s(t')$.*

Proof. In $\mathcal{M} \cup \mathcal{M}_t$, the variables and the constraints of $\mathcal{M}$ are left unchanged; only additional model variables are introduced with additional constraints 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 $n_1, \ldots, n_k$ 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 *reified constraint* is posted in the model, for translating the leaf c:

$$\ell(n_1) = p_1 \wedge \cdots \wedge \ell(n_k) = p_k \Rightarrow c$$

Let $X \in V(\mathcal{M}_t)$ be one of the variables that label or-nodes. The domain of X will be $\texttt{0..}\max\{\mathrm{w}(n) - 1 \mid \ell(n) = X\}$ where $\mathrm{w}(n)$ 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:

$$\ell(n_1) = p_1 \wedge \cdots \wedge \ell(n_{k-1}) = p_{k-1} \Rightarrow \ell(n_k) < \mathrm{w}(k)$$

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

$$\left(\bigwedge_{\substack{(n_i, p_i)_i \\ \ell(n_k) = X}} \ell(n_1) \neq p_1 \vee \cdots \vee \ell(n_k) \neq p_k \right) \Rightarrow X = 0$$

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. To reproduce the search procedure of $(\mathcal{M}, t)$, it is sufficient for t' to be a standard labeling and/or tree where the variables are ordered according to the ordering of the corresponding nodes in t. The value selection strategy fixes the order in which the sub-branches are explored. For t', it can be the standard left-most selection strategy of labeling values.

By construction, $(\mathcal{M} \cup \mathcal{M}_t, t')$ thus explores the same search tree as $(\mathcal{M}, t)$. □

Although not mandatory for correctness, the following optimization has been measured to give significant performance improvements, by reducing the number of generated constraints.

Proposition 1. *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 all other branches of n_k, the constraints corresponding to the leaves c and $\neg c$ are logically equivalent to the* *following reified constraint*

$$\ell(n_1) = p_1 \wedge \cdots \wedge \ell(n_{k-1}) = p_{k-1} \Rightarrow (\ell(n_k) = p_k \Leftrightarrow c)\,.$$

That is to say, the implication on the leaf c may be replaced by an equivalence and the constraints corresponding to the leaves $\neg c$ are not posted.

4. Extending MiniZinc with Horn Clauses

MiniZinc [18] is a constraint modeling language which allows the writing of search annotations dedicated to CP solvers. In our approach this is not needed. Instead, we extend MiniZinc with Horn clauses for specifying search procedures declaratively, and use the previous theorem for transforming ClpZinc search constraint models into MiniZinc constraint models using the standard labeling strategy.

4.1 The Language ClpZinc

Given a constraint system $\mathcal{X}$ (e.g. finite domain constraints) and a constraint model over $\mathcal{X}$, 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 MiniZinc data structures makes the language look familiar to Prolog users and other constraint logic programming (CLP) systems. Similarly, we fix the strategy as depth-first and left-to-right.

The language ClpZinc is an extension of MiniZinc where the item `solve satisfy;` in models is replaced by a goal of the form "`:-` *goal*.", and where user-defined predicates are defined by Horn clauses of the form "`p(`t_1`,...,`t_n`) :-` *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 = 0;
constraint X1 = 1 -> x = 1;
constraint X1 = 2 -> x = 2;
constraint X1 = 3 -> x = 3;
constraint X1 = 4 -> x = 4;
constraint X1 = 5 -> x = 5;
solve :: seq_search([
    int_search([X1], input_order, indomain_min, complete)
  ]) satisfy;
output [show(x)];
```

Definition 5. *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(`t_1`,...,`t_n`) :-` goal. *where t_1 and t_n are terms and* goal *is a ClpZinc goal. The goal part can be omitted: "*`p(`t_1`,...,`t_n`).`*" is a shorthand for "*`p(`t_1`,...,`t_n`) :- 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`, ...), numbers, or compound terms of the form `p(`t_1`,...,`t_n`)` where t_1, ..., and t_n are terms. Model variables are a special case of compound terms, either atomic (`a`, `b`, ...) or array accessors (`x[I,J]`). MiniZinc arrays have been unified with Prolog-like lists to ease their enumeration in search strategies.

In 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`, ...) 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 $\mathcal{X}$ of $\mathcal{H}$ are unified.

Arithmetic expressions are also extended for accessing the indexicals of the model variables. For instance, the goal `M = 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 MiniZinc* `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* `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 splitting strategy introduced in [21] for solving Korf's packing problem [14], by making a preliminary coarse-grained filtering of the variable domains.

Example 3 (Interval splitting)**.** *The* `interval_splitting/4` *predicate, 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 = 0 <-> x < X1 + 2;
constraint X3 = 1 -> (X4 = 0 <-> x < X2 + 2);
constraint X3 = 0 -> 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;
```

4.2 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 [4]. The description of these techniques is however beyond the scope of this paper.

Given a constraint system $\mathcal{X}$, the partial evaluation of a 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 left-to-right evaluation of Horn clauses, the and/or trees will be traversed in a similar left-to-right fashion in our examples, but any other traversal order can be treated similarly (see for example Figure 3 where the branches of the and-nodes should be executed from left to right). Note that the transformation itself does not follow a strict DFS and does not either rely on the classical CSLD resolution, which would lead to a pure or-tree. It is rather defined as a continuation-based generation of the and/or tree. The precise definition is given below, where s is the initially empty partial evaluation store built along the transformation and $\Box$ denotes a *hole* around which the context is built and labeled by such a store. The unfolding of predicate calls and resolution of arithmetic tests is left out for readability reasons.

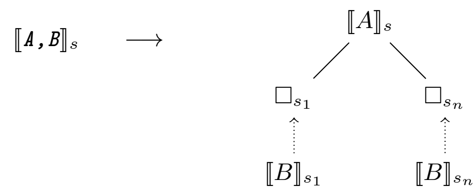
Definition 6. *The transformation* $[\![\bullet]\!]_s$: goal $\rightarrow$ search-tree*, in the partial evaluation store* $s \in \mathcal{H}$*, is defined by structural induction on the goal by*

$$[\![\mathtt{true}]\!]_s \longrightarrow \Box_s$$

$$[\![\mathtt{false}]\!]_s \longrightarrow \bot$$

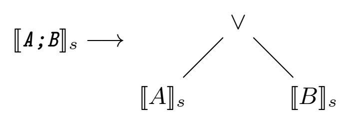
$$[\![\mathtt{X = v}]\!]_s \longrightarrow \Box_{s\wedge(\mathtt{X = v})}$$

where `c` *is a constraint or a search annotation*

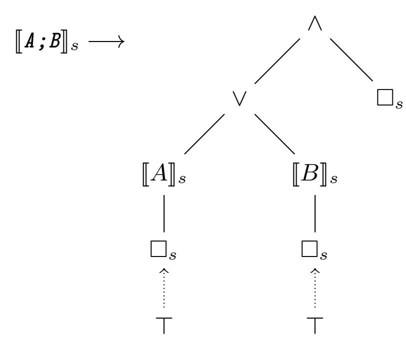


all $\Box_{s_i}$ *of* $[\![A]\!]_s$ *are filled with* $[\![B]\!]_{s_i}$
i.e., $[\![A]\!]_s[\forall i,\ [\![B]\!]_{s_i}/\Box_{s_i}]$

- *if* A *or* B *changes the store,* i.e., $\exists s' \neq s,\ \Box_{s'} \in [\![A]\!]_s$ *or* $[\![B]\!]_s$*:*



- *if neither* A *nor* B *changes the store:*



the leftmost leaf is $[\![A]\!]_s[\top/\Box_s]$ *and its sibling* $[\![B]\!]_s[\top/\Box_s]$

The associativity of the nodes is enforced during the transformation, allowing *n-ary* nodes.

4.3 Examples of Transformation

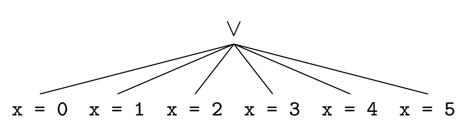


Figure 4. And/Or tree of Example 1 for labeling.

In Figure 4, the variable `X1` is assigned to the root node, with the domain `0..5` corresponding to the arity of the node. As shown in the MiniZinc 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 `X1`. Therefore, when the search annotation `int_search` enumerates the possible values of `X1`, these guarded constraints are successively enabled for exploring the different branches of the tree.

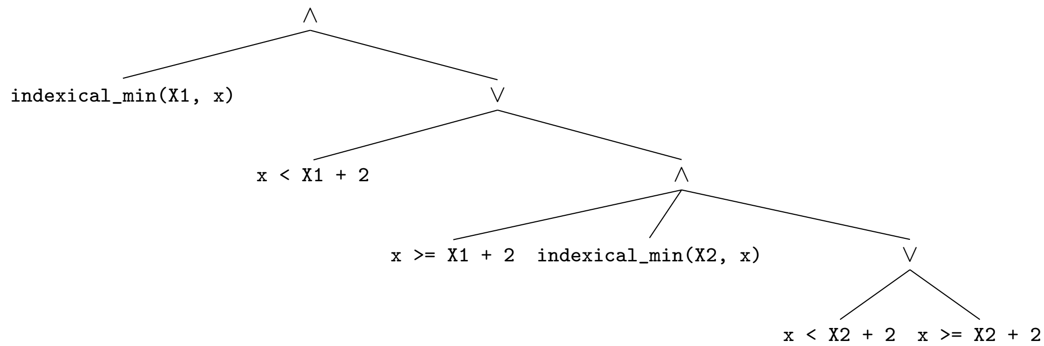


Figure 3. And/Or tree of Example 3 for interval splitting.

As shown in Figure 4, 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 5 shows a simple example of duplication with partial instantiation of the bounding constraint `Min <= x, x <= Max`.

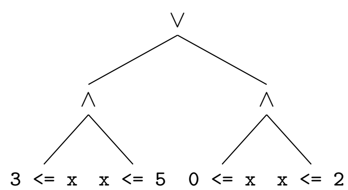


Figure 5. And/Or tree for the ClpZinc goal

```
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 6 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.

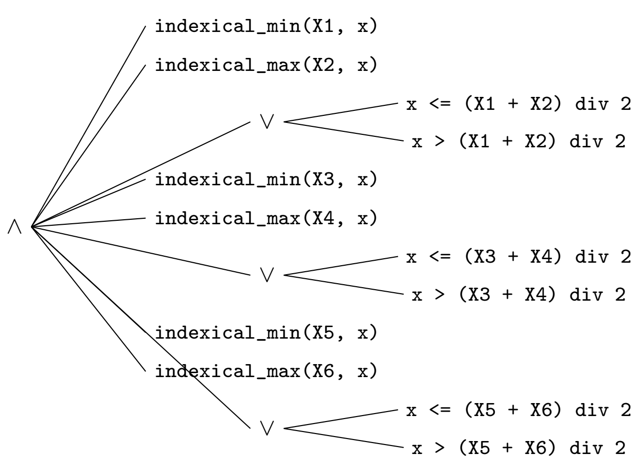


Figure 6. And/Or tree of Example 2 for dichotomic search.

5. Computation Results on Korf's Square Packing Benchmark

In this section, we consider Korf's Optimal Rectangle Square Packing problem, *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. Figure 7 depicts one provably optimal placement for $n = 26$.

Helmut Simonis and Barry O'Sullivan proposed in [21] a constraint logic program in SICStus Prolog [5], using global constraints and a complex dynamic search strategy for solving that problem, beating the best results obtained so far at that time.

Their constaint model involves three global constraints: a geometric non-overlapping constraint over the squares, and two redundant constraints that express in each dimension that the sum of lengths of squares placed at every abscissa (resp. ordinate) should not exceed the height (resp. the width) of the enclosing rectangle. This is expressed by two "cumulative" constraints that are usually used in resource allocation and scheduling [6].

$$\begin{cases} \text{non-overlap}(\{(x_i, y_i, i \times i) \mid 1 \leq i \leq n\}) \\ \text{cumulative}(\forall 1 \leq x \leq w, \sum\{i \mid x_i \leq x \leq x_i + i\} \leq h) \\ \text{cumulative}(\forall 1 \leq y \leq h, \sum\{i \mid y_i \leq y \leq y_i + i\} \leq w) \end{cases}$$

This model which relates abscissas x_i with the height h and ordinates y_i with the width w gives surprisingly good experimental results with a strategy that first fully enumerates in one dimension, *e.g.* x_i, and then fully enumerates in the other one, y_i. The enumeration strategy of Simonis and O'Sullivan first makes a coarse-grained placement which splits each dimension into intervals up to one third of the square widths, from the biggest square to the smallest one, and then the placement is refined by dichotomy, still from the biggest square to the smallest one.

First, the constraint 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(
```

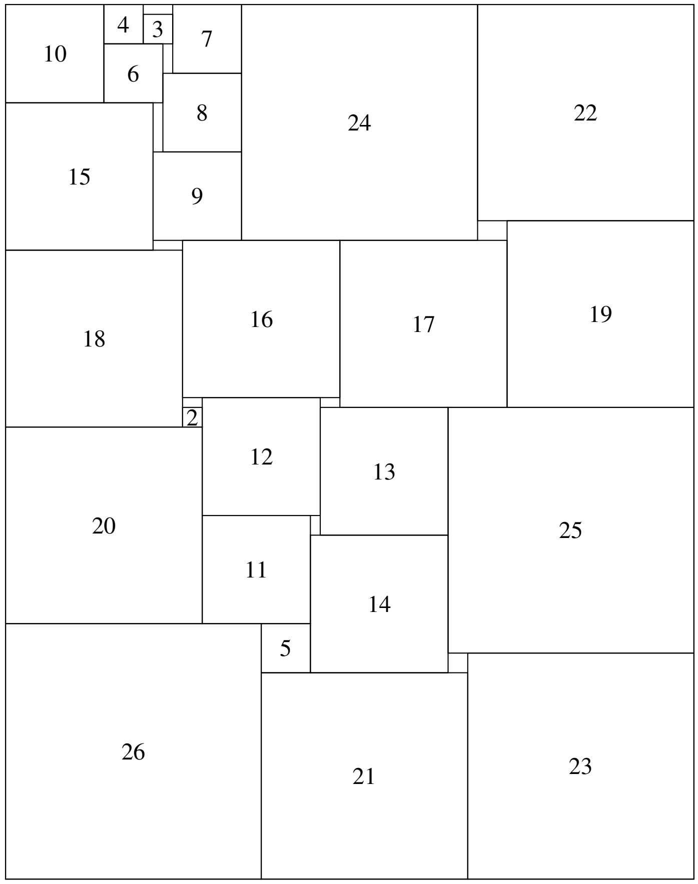


Figure 7. One optimal solution to Korf's problem for packing 26 squares of size 1 to 26 in an enclosing rectangle of least area. The 1×1 square is omitted since it can be put in any interleaving between squares.

```
   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 [21] 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 `max_size` $\times$ `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 [21]:

```
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 [21] 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
   ).
```

where `reverse` is implemented as usual with Horn clauses.

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-i] | i in 1..n-1] ++ [y[n-i] | i in 1..n-1],
         input_order, indomain_split, complete
      )
   ]) satisfy;
```

To measure the overhead of ClpZinc over MiniZinc, 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).
```

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 using either Choco or SICStus solvers, and of the original SICStus-Prolog program of [21], all of them running on Intel®Xeon®CPU E5-1620 0 @ 3.60GHz machines. For the number of squares to pack, $16 \leq n \leq 19$, the maximum length of the enclosing rectangle is `max_size`=80; for $20 \leq n \leq 24$, `max_size`=100; for $25 \leq n \leq 26$, `max_size`=150. As shown in Table 1, the overhead introduced by the reification of the search procedure is quite reasonable, averaging a two-fold and at reaching at most a three-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 [21]. Note that, although the number of generated constraints is much larger than in the original model, this number mostly depends on `max_size` and appears to have little impact on the observed overhead, which remains quite constant.

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

6. 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) [12] and Symmetry Breaking During Search (SBDS) [10] as strategy transformers for labeling or dichotomic search for instance. This technique is closely related to the monadic approach of strategy transformers presented in [19]. The main difference, outside of purely syntactic choices, is that the monadic transformations described in [19] 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.

6.1 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;
constraint x = 0;
constraint y = 0;
constraint x != y;
solve :: seq_search([
    int_search(a, input_order, indomain_min, complete)
  ]) satisfy;
n = 1000;
```

6.2 Symmetry Breaking During Search

Symmetry Breaking During Search [2, 10] 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 a given 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 goal by meta-interpretation.

7. 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

n	Choco 3			SICStus		number of generated MiniZinc constraints
	dichotomic `indomain_split`	dichotomic ClpZinc	interval split then dichotomic ClpZinc	interval split then dichotomic ClpZinc	interval split then dichotomic Original	
16	9	14	0.853	0.710	0.340	1199
17	16	22	0.982	0.450	0.250	1249
18	422	570	8	9	5	1299
19	785	1051	7	12	4	1349
20			13	17	9	1573
21			43	88	32	1619
22			209	304	154	1859
23			1341	2104	999	1913
24			2313	3433	1482	1959
25			29202	33983	10663	3039
26			142702	180231	62180	3109

Table 1. Solving times in seconds for Korf's packing problem with Simonis and O'Sullivan's search strategy implemented in ClpZinc with Choco 3 and SICStus prolog as solvers, compared to the original SICStus prolog program of [21].

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 assignation to `id` is skipped outside the computation branch that involves this assignation. 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 labeling `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.

8. Encoding Search Combinators

Search combinators [20] introduce a domain-specific language for modeling search. We recall the primitive constructions of this domain-specific language in Table 8. Search combinators are known to go beyond the conjunctions and disjunctions of CLP goals, as it is mentioned in the related work section of [20]. However, we show that these combinators are straightforward to express using Horn clauses through meta-interpretation, thus can be expressed through the above transformation as labeling in pure CSPs. The only restriction is on `restart`: since the unfolding of Horn clauses into search trees should terminate, the number of iteration should be bounded statically. We only illustrate some encodings.

The `prune` combinator is encoded as a failure at constraint level, as in section 7.

```
prune :-
    store(prune, 1),
    domain([A, B, C], 0, 1),
    A != B, B != C, A != C.
```

The global state `prune` allows `portfolio` to implement the exhaustiveness check: by definition, the search tree is exhaustive if no call to `prune` are performed during its exploration.

```
portfolio([]) :-
    prune.
portfolio([H | T]) :-
    retrieve(prune, Former_prune),
    store(prune, 0),
    H,
    retrieve(prune, New_prune),
    store(prune, Former_prune),
    (
        New_prune = 0
    ;
        New_prune = 1,
        portfolio(T)
    ).
```

`let` and `assign` rely on a global state. `new_atom` is a primitive that generates a fresh Herbrand function symbol.

```
let(X, V, S) :-
    new_atom(X),
    store(X, V),
```

s ::= `prune`
prunes the current search branch (*i.e.*, fails)
| `base_search(`*vars*`,`*var-select*`,`*domain-split*`)`
search annotation
| `let(`*x*`,`*initial-value*`,`*s*`)`
introduces a new variable (*i.e.*, global state) *x* in *s*
| `assign(`*x*`,`*new-value*`)`
changes the value of the global state *x*
| `post(`*c*`,`*s*`)`
posts the constraint *c* at every choice-point during *s*
| `ifthenelse(`*cond*`,`s_1`,`s_2`)`
substitutes in s_1 every subtree where *c* is false by s_2
| `and([`s_1`,`s_2`,...,`s_n`])`
performs s_1, s_2, up to s_n while they succeed,
and fails otherwise
| `or([`s_1`,`s_2`,...,`s_n`])`
if s_1 succeeds then succeeds, otherwise if s_2 succeeds,
then succeeds, otherwise...
| `portfolio([`s_1`,`s_2`,...,`s_n`])`
performs s_1, s_2, up to s_n until one of them is
exhaustive (*i.e.*, does not perform `prune`)
| `restart([`*cond*`,`*s*`])`
restarts *s* as long as *cond* holds

Table 2. Primitive search combinators from [20].

```
    S.

assign(X, V) :-
    store(X, V).
```

`ifthenelse` and `post` interleaves the exploration of the search tree with the test of the condition by meta-interpretation. We illustrate this meta-interpretation on `post`.

```
post(C, (A, B)) :-
    post(C, A),
    post(C, B).

post(C, (A ; B)) :-
    (
        C, A
    ;
        C, B
    ).

post(C, B) :-
    builtin(B),
    C,
    B.
```

Finally, note that even if the `first_fail` variable selection strategy (or a similar one) is not available as a built-in in the underlying solver, it is once again quite straightforward to implement in ClpZinc. Remark that, due to the lack of access to indexicals, such an encoding is not possibles using search combinators which are limited to what `base_search` makes available. The `first_fail` variable selection strategy selects the variable with the smallest domain among the variables that are not already instantiated (*i.e.*, whose domain is not reduced to a singleton). If all variables are instantiated, the predicate returns by convention the first variable, so that there is no failure when iterating this predicate once per model variable, which ensures complete instantiation.

```
first_fail(Vars, X) :-
   select(X, Vars, Other_vars),
   CardX = card(X),
   CardX > 1,
   check_card_greater_than(Other_vars, CardX).

first_fail(Vars, X) :-
   fully_instantiated(Vars),
   Vars = [H | _].

check_card_greater_than([], _).

check_card_greater_than([H | T], Min) :-
   CardH = card(H),
   (CardH < Min -> CardH = 1),
   check_card_greater_than(T, Min).

fully_instantiated([]).

fully_instantiated([H | T]) :-
   card(H) = 1,
   fully_instantiated(T).
```

9. 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 arithmetic and reified constraints, and implemented in constraint solvers without sophisticated support for search. This concept of search by constraint propagation makes search procedures declarative and leads to a novel implementation technique of search with wide potential use in constraint modeling languages.

On the complex dynamic strategy used for solving Korf's benchmark for square packing, we have shown that the implementation overhead of this declarative implementation technique is limited to a factor 3 due to the management of reified constraints. This overhead can be measured on different constraint solvers without 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 using an extension of the MiniZinc modeling language with Horn clauses, called ClpZinc, which provides recursion to ease the definition of search strategies and compiles to MiniZinc. Under termination assumptions the ClpZinc compiler expands a goal in a MiniZinc model which implements the search strategy with reified constraints. 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 constraint solvers with limited built-in search strategies. We have added the needed indexicals to the FlatZinc parser of some solvers (Choco [8], JaCoP [15], SICStus [1], Gecode [7], or-tools [11]) and encourage all solver developers to do so in order to tackle these new challenging problems for the MiniZinc community. For instance, Korf's packing problem with the complex strategy of [21] 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 [21] 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 constraint solvers. Furthermore, work on hard combinatorial problems with dedicated heuristics could now become independent of particular solvers through the declarative modeling of the search strategy in our approach.

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 [17] 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 ANR-11-BS02-0005 Net-WMS-2 project, and to the partners of this project and particularly to Philippe Morignot for discussions on these topics.

References

[1] S. AB. Sicstus 4.2.3, 2012. URL `http://sicstus.sics.se/`.

[2] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In J. Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999. ISBN 3-540-66626-5. . URL `http://dblp.uni-trier.de/db/conf/cp/cp99.html{\#}BackofenW99`.

[3] Y. Bekkers and P. Tarau. Monadic constructs for logic programming. In J. Lloyd, editor, *Logic Programming*, pages 51–65. MIT Press, Cambridge, MA,, 1995.

[4] 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.

[5] M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. URL `http://www.sics.se/sicstus/`. ISBN 91-630-3648-7.

[6] Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *JICSLP*, volume 96, pages 369–383, 1996.

[7] T. G. Community. Gecode 4.2.1, 2013. URL `http://www.gecode.org/`.

[8] É. des Mines de Nantes. Choco 3, 2014. URL `http://www.emn.fr/z-info/choco-solver/index.php?page=choco-3`.

[9] A. M. Frisch, W. Harvey, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[10] I. P. Gent and B. Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, pages 599–603, 1999.

[11] Google. or-tools, 2014. URL `http://or-tools.googlecode.com/`.

[12] W. D. Harvey and M. 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. ISBN 1-55860-363-8, 978-1-558-60363-9.

[13] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, Jan. 1987.

[14] R. E. Korf. Optimal rectangle packing: Initial results. In E. Giunchiglia, N. Muscettola, and D. S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003. ISBN 1-57735-187-8. URL `http://dblp.uni-trier.de/db/conf/aips/icaps2003.html{\#}Korf03`.

[15] K. Kuchcinski and R. Szymanek. Jacop 4.0.0, 2013. URL `http://jacop.osolpro.com/`.

[16] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.

[17] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 16(9):357–391, 2009.

[18] R. Rafeh, K. Marriott, M. G. de la Banda, N. Nethercote, and M. Wallace. Adding search to zinc. In *CP*, pages 624–629, 2008.

[19] T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663, 2009.

[20] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. J. Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.

[21] H. Simonis and B. O'Sullivan. Search strategies for rectangle packing. In P. J. Stuckey, editor, *Proceedings of CP'08*, volume 5202 of *LNCS*, pages 52–66. Springer-Verlag, 2008.

[22] P. Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.