Search as Constraint Satisfaction

Thierry Martinez
Joint work with François Fages, Philippe Morignot and Sylvain Soliman

(submitted to CP’2014)
The two components of Constraint Programming

Constraint Programming = Model + Strategy
The model

- Model: CSP, “Constraint Solving Problem”
  - $\mathcal{X}$ - a set of variables
  - $\mathcal{D}$ - a family of domains
  - $\mathcal{C}$ - a set of constraints $\sim$ relations over variables

One successful language: Zinc;

and many solvers: Choco, Gecode, JaCoP, or-tools, ...
Zinc model for Korf’s Square Packing Problem

Optimization problem: given an integer $n$, find a rectangle of minimal area containing the $n$ squares from $1 \times 1$ to $n \times n$.

```zinc
include "cumulative.mzn";
include "diffn.mzn";

int: n; 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 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 w * h = area;

solve :: seq_search(
    int_search([area, w], input_order, indomain_min, complete),
    int_search(x ++ y, first_fail, indomain_min, complete)
) satisfy;
```
But sometimes models are not enough...

Complete method:
The search space to explore can be large: need for a strategy
The strategy

- **Strategy**: “a search procedure”, “an enumeration strategy”, “a decision tree”, ...

Enumeration strategy = variable selection + value selection

~ Zinc annotations

int_search([x,y,z], first_fail, indomain_split, complete)

Other search procedures: interval splitting, shaving, dynamic symmetry breaking, ...

Implementation of general strategies is solver dependent.
The clp2zinc Theorem

Any strategy expressible as the traversal of an and/or tree can be internalized as auxiliary constraints with a fixed enumeration strategy.

$$(M, t) \rightarrow (M + M_t, t')$$

The transformation is constructive:
http://lifeware.inria.fr/~tmartine/clp2zinc/
clp2zinc

What is it?

What does it do?

How does it work?
What is it?

- A language to describe and/or trees: ClpZinc (an implementation of CLP($\mathcal{X} + \mathcal{H}$))

- A transformation from and/or trees to constraints (and from a tree traversal to a fixed enumeration strategy)
A language to describe and/or trees

\[ \land \quad \text{indexical}_\text{min}(X_1, x) \quad \text{indexical}_\text{max}(X_2, x) \]
\[ \lor \quad x \leq (X_1 + X_2) \div 2 \quad x > (X_1 + X_2) \div 2 \]
\[ \text{indexical}_\text{min}(X_3, x) \quad \text{indexical}_\text{max}(X_4, x) \]
\[ \lor \quad x \leq (X_3 + X_4) \div 2 \quad x > (X_3 + X_4) \div 2 \]
\[ \text{indexical}_\text{min}(X_5, x) \quad \text{indexical}_\text{max}(X_6, x) \]
\[ \lor \quad x \leq (X_5 + X_6) \div 2 \quad x > (X_5 + X_6) \div 2 \]
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).

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.

\(\mathcal{X} + \mathcal{H}\): a unique notion of equality “=” that operates on \(\mathcal{X}\) and \(\mathcal{H}\).
(there is no longer “#=”,” “is”...)
MiniZinc generated code for dichotomy

```plaintext
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(X3, 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;
```
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).
Interval splitting: and-or tree

\[\text{indexical\_min}(X_1, x) \land (x < X_1 + 2 \lor (x \geq X_1 + 2 \land \text{indexical\_min}(X_2, x)) \lor (x < X_2 + 2 \land x \geq X_2 + 2))\]
Interval splitting: generated code

```cpp
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;
```
What does it do?

- A declarative strategy modeling language targeting many solvers: Choco, JaCoP, Gecode, or-tools, SICStus, ...

- Solve Korf’s Square Packing Problem (small overhead, 2×/3× slower) Useful for comparing solvers!

- Support Strategy Transformers via Meta-Interpretation

- Speed-up Thanks To Constraint Propagation
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).
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).
Speed-up with LDS!

\[ \text{var 0..1: x; } \]
\[ \text{var 0..1: y; } \]
\[ \text{array[0..n] of var 0..1: a; } \]

\[ \text{:- int_search(a, input_order, indomain_min, complete), } \]
\[ \text{lds(((x = 0; x = 1), (y = 0; y = 1)), 0), x != y. } \]
Symmetry-Breaking During Search

\( \lor \)

\[ c \]

\[ \sigma(\neg c) \]

\[
\text{sbd} \text{ds}(\text{top, } \_).
\text{sbd} \text{ds}(\text{or}(A, B), \text{Path}) :-
\begin{align*}
( & A = \text{constraint}(C, A0), \\
& ( & C, \text{sbd} \text{ds}(A, [C | \text{Path}] ) \\
& ; \text{cut_symmetry}(C, \text{Path}), \text{sbd} \text{ds}(B, \text{Path})) \\
& ; A \neq \text{constraint}(\_, \_), \\
& (\text{sbd} \text{ds}(A, \text{Path}); \text{sbd} \text{ds}(B, \text{Path}))).
\end{align*}
\]

\[
\text{sbd} \text{ds}(\text{constraint}(C, T), \text{Path}) :- C, \text{sbd} \text{ds}(T, [C | \text{Path}]).
\]

\[
:\text{search_tree}(\text{labeling_list}(\text{queens, 1, n}), T), \text{sbd} \text{ds}(T, []).
\]
Beyond And-Or Trees

\[ X = s_0 \]

\[ X > s_0 \]

\[ X = s_1 \]

\[ X > s_1 \]

\[ \ldots \]

\[ X > s_n \]

\[ \times \]

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 \neq B \land B \neq C \land A \neq 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).
How does it work?

ClpZinc $\rightarrow$ and/or trees $\rightarrow$ MiniZinc
and/or trees $\rightarrow$ MiniZinc

```plaintext
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)];
```
Many levels, equivalences, rejects...
ClpZinc $\rightarrow$ and/or trees

Prolog evaluation makes or-trees:

$$(A ; B), C \rightarrow (A, C) ; (B, C)$$

$\forall x > 2$

$\forall x = 5$

$\forall x = 4$

$\forall x = 3$

$\forall x = 2$

$\forall x = 1$

$\forall x = 0$
From goals to contexts

$s$ is a store of $\mathcal{H}$ constraints

\[
\begin{align*}
\square_s & \quad [\text{true}]_s \quad \rightarrow \\
\bot & \quad [\text{false}]_s \quad \rightarrow \\
\square_s + (X = v) & \quad [X = v]_s \quad \rightarrow \\
\land & \quad [c]_s \quad \rightarrow \\
\square_s & \quad [A, B]_s \quad \rightarrow \\
\square_{s_1} & \quad [B]_{s_1} \\
\square_{s_n} & \quad [B]_{s_n}
\end{align*}
\]
Contexts for choice-points

- if $A$ or $B$ change the store:

  $$[[A;B]]_s \rightarrow \lor \begin{cases} \lbrack A \rbrack_s \\ \lbrack B \rbrack_s \end{cases}$$

- if $A$ and $B$ do not change the store:

  $$[[A;B]]_s \rightarrow \land \begin{cases} \lbrack A \rbrack_s \\ \lbrack B \rbrack_s \end{cases} \lor \Box_s$$
Conclusions...

▶ CSP + reified constraints has the expressive power to internalize search strategies.

▶ CLP as a solver-independent strategy modeling language.

▶ Seamless integration between Zinc and CLP

... and perspectives

▶ Search strategies are supposed to terminate...

▶ Can we really *program* in a CSP? Can we bootstrap ClpZinc with a solver over the domain of CSPs and a clever search strategy?

▶ Beyond And-Or Trees... but is it beyond any logic?