Constraint Logic Programming

Sylvain Soliman
Sylvain.Soliman@inria.fr

Project-Team LIFEWARE

MPRI 2.35.1 Course – September–November 2017

Part I: CLP - Introduction and Logical Background

1. The Constraint Programming paradigm
2. Examples and Applications
3. First Order Logic
4. Models
5. Logical Theories

Part II: Constraint Logic Programs

6. Constraint Languages
7. CLP($\mathcal{X}$)
8. CLP($\mathcal{H}$)
9. CLP($\mathcal{R}, \mathcal{FD}, \mathcal{B}$)

Part III: CLP - Operational and Fixpoint Semantics

10 Operational Semantics

11 Fixpoint Semantics

12 Program Analysis

Full abstraction

Theorem 1 ([JL87popl])

$T_P^{\mathcal{X}} \uparrow \omega = O_{gs}(P)$

$T_P^{\mathcal{X}} \uparrow \omega \subset O_{gs}(P)$ is proved by induction on the powers n of $T_P^{\mathcal{X}}$.
$n = 0$, i.e., $\emptyset$, is trivial. Let $A\rho \in T_P^{\mathcal{X}} \uparrow n$, there exists a rule $(A \leftarrow c|A_1, \ldots, A_n) \in P$, s.t. $\{A_1\rho, \ldots, A_n\rho\} \subset T_P^{\mathcal{X}} \uparrow n-1$ and $\mathcal{X} \models c\rho$. By induction $\{A_1\rho, \ldots, A_n\rho\} \subset O_{gs}(P)$. By definition of O_{gs} and $\wedge$-compositionality. we get $A\rho \in O_{gs}(P)$.
$O_{gs}(P) \subset T_P^{\mathcal{X}} \uparrow \omega$ is proved by induction on the length of derivations. Successes with derivation of length 0 are ground facts in $T_P^{\mathcal{X}} \uparrow 1$.
Let $A\rho \in O_{gs}(P)$ with a derivation of length n. By definition of O_{gs} there exists $(A \leftarrow c|A_1, \ldots, A_n) \in P$ s.t. $\{A_1\rho, \ldots, A_n\rho\} \subset O_{gs}(P)$ and $\mathcal{X} \models c\rho$. By induction $\{A_1\rho, \ldots, A_n\rho\} \subset T_P^{\mathcal{X}} \uparrow \omega$. Hence by definition of $T_P^{\mathcal{X}}$ we get $A\rho \in T_P^{\mathcal{X}} \uparrow \omega$.

Part IV: Logical Semantics

13 Logical Semantics of CLP($\mathcal{X}$)

14 Automated Deduction

15 CLP(λ)

16 Negation as Failure

Part V: Constraint Solving

17 Solving by Rewriting

18 Solving by Domain Reduction

Part VI: Practical CLP Programming

19 CLP implementation, the WAM

20 Optimizing CLP

21 Symmetries

22 Symmetry Breaking During Search

23 Detecting Symmetries

SBDS

In 99, [BW99cp] proposed a completely different symmetry breaking technique, **Symmetry Breaking During Search** (SBDS).

It overcomes the main drawback of static symmetry breaking:

SBDS

In 99, [BW99cp] proposed a completely different symmetry breaking technique, **Symmetry Breaking During Search** (SBDS).

It overcomes the main drawback of static symmetry breaking: the choice of the representative element in each class of solutions

SBDS

In 99, [BW99cp] proposed a completely different symmetry breaking technique, **Symmetry Breaking During Search** (SBDS).

It overcomes the main drawback of static symmetry breaking: the choice of the representative element in each class of solutions is forced

breaking all trials at improving performance by

SBDS

In 99, [BW99cp] proposed a completely different symmetry breaking technique, **Symmetry Breaking During Search** (SBDS).

It overcomes the main drawback of static symmetry breaking: the choice of the representative element in each class of solutions is forced

breaking all trials at improving performance by clever search heuristics

Symmetric Constraints

Consider a set Σ of symmetries, such that for any constraint *c* and all $\sigma \in \Sigma$ one can find a constraint $\sigma(c)$ corresponding to the symmetric of *c*
$\mathcal{X} \models \sigma(c)\rho \Leftrightarrow c\sigma(\rho)$

For example, if σ is the value symmetry that turns *v* into $N - v$ we have $\sigma($`X = v`$)$ is `X = (N - v)`

We can now define a technique for removing symmetries adding constraints when choice-points are explored, *à la* branch and bound.

Enumerating Solutions

The general method of enumeration of solutions is, at each choice-point, to add

- on one branch the constraint c assigning a value to a variable;
- on the other branch the negation of this constraint $\neg c$

SBDS adds supplementary constraints on the second branch:

supposing a partial assignment $\mathcal{A}$ at the choice-point,
for all $\sigma \in \Sigma$ such that $\sigma(\mathcal{A}) = \mathcal{A}$ one adds $\sigma(\neg c)$.

Example

Consider the 4-queens problem over $X_1, X_2, X_3, X_4 \in \{1, 2, 3, 4\}$

with a single (value-)symmetry: $v \mapsto 5 - v$

suppose that at the top of the search tree the leftmost branch corresponds to $X_1 = 1$

when backtracking at the top, the next branch to explore will correspond to the constraint:

$$X_1 \neq 1 \wedge X_1 \neq 4$$

Unicity

Theorem 2 (Non-symmetric Solutions)

If ρ_1 and ρ_2 are two solutions obtained by SBDS, then

$$\forall \sigma \in \Sigma \qquad \sigma(\rho_1) \neq \rho_2$$

Proof.

Suppose that $\sigma_0(\rho_1) = \rho_2$ for some σ_0
let $\mathcal{A}$ be the partial assignment at the choice-point that differentiates the ρ_1 and ρ_2 branches, and *c* the constraint added on the ρ_1 branch there.
We have $\sigma_0(\mathcal{A}) = \mathcal{A}$
since both are solutions, we get that *c* is true in ρ_1
and that $\sigma_0(\neg c)$ is true in ρ_2 i.e., $\neg c$ is true in ρ_1
$\Rightarrow$ contradiction $\square$

About Partial Assignments

If one adds systematically $\sigma(\neg c)$ even when $\sigma(\mathcal{A}) \neq \mathcal{A}$

About Partial Assignments

If one adds systematically $\sigma(\neg c)$ even when $\sigma(\mathcal{A}) \neq \mathcal{A}$ one loses solutions!

Example 3

Consider again the 4 queens problem,
at some point we explore the branch $X_1 = 2$ and then $X_2 = 1$

About Partial Assignments

If one adds systematically $\sigma(\neg c)$ even when $\sigma(\mathcal{A}) \neq \mathcal{A}$
one loses solutions!

Example 3

Consider again the 4 queens problem,
at some point we explore the branch $X_1 = 2$ and then $X_2 = 1$
$\Rightarrow$ failure

About Partial Assignments

If one adds systematically $\sigma(\neg c)$ even when $\sigma(\mathcal{A}) \neq \mathcal{A}$
one loses solutions!

Example 3

Consider again the 4 queens problem,
at some point we explore the branch $X_1 = 2$ and then $X_2 = 1$
$\Rightarrow$ failure $\Rightarrow$ $X_2 \neq 4$
we never find any solution…

Conversely, new local symmetries might appear in some partial assignments (the overhead of handling those is usually not worth it).

Detecting Symmetries

[GHK02cp] show that constraint symmetries such as those considered for SDBS form a group

they link CSPs (in ECLiPSe) with the GAP computational abstract algebra system

many symmetries (even local ones) can be detected automatically

remains costly and not much used…

Part VII

More Constraint Programming

Part VII: More Constraint Programming

24 Typing CLP

25 CHR

Prescriptive vs. Descriptive Typing

"Well typed programs never go wrong"

Descriptive type systems try to upper-approximate the denotation (i.e., success set) of a program.

Subject reduction ensures that well-typedness is conserved during the execution.

`append(X, [4, 5], [])` has no success…
`append([], X, X)` has a success, whatever `X`

What should the type of `append` be?

Prescriptive Type Systems

Defined by the user to express the intended use of function and predicate symbols in programs.

Orthogonal to the question of the feasibility of type inference.

Subject reduction becomes a verification of the consistency of the type system w.r.t. the execution model (in our case,

Prescriptive Type Systems

Defined by the user to express the intended use of function and predicate symbols in programs.

Orthogonal to the question of the feasibility of type inference.

Subject reduction becomes a verification of the consistency of the type system w.r.t. the execution model (in our case, the CSLD resolution).

Meta-programming

```
functor(X, F, N).
call(G).
setof(X, G, L).
```

Even *parametric polymorphism*, introduced by Damas-Milner for ML and adapted to logic programming by [MycroftOkeefe84ai] is not enough.

Meta-programming

```
functor(X, F, N).
call(G).
setof(X, G, L).
```

Even *parametric polymorphism*, introduced by Damas-Milner for ML and adapted to logic programming by [MycroftOkeefe84ai] is not enough.

Subtyping is!

$$\begin{aligned} \mathit{list}(\alpha) &\leq \mathit{term} \\ \mathit{pred} &\leq \mathit{term} \\ \mathit{list}(\alpha) &\not\leq \mathit{pred} \end{aligned}$$

[CF01tplp]

they obtain subject reduction w.r.t. substitutions and CSLD resolution with p.o.-terms and covariant constructors.

type checking amounts to solving left-linear and acyclic inequalities $\Rightarrow$ linear algorithm
type inference is slightly harder (non-left-linear inequalities appear)

the whole SICStus library was checked (around 600 predicates, quite similar to SWI) with type declarations only for the about 100 built-ins.
Inferred types were exact in vast majority; a few errors were also detected.

Constraint Handling Rules (CHR)

- Constraint programming language for Computational Logic
- Created by Thom Frühwirth in 1991
- Multi-headed guarded committed-choice rules transform multi-set of constraints until exhaustion
- Ideal for concise executable specifications and rapid prototyping
- Any-time (approximation), on-line (incrementality), concurrent algorithms for free.
- Logical and operational semantics coincide strongly
- High-level supports program analysis and transformation: Confluence/completion, operational equivalence, termination/time complexity, correctness…

Syntax

Simplification rule:
$H \Leftrightarrow C \mid B$

Propagation rule:
$H \Rightarrow C \mid B$

- H: non-empty conjunction of CHR constraints
- C: conjunction of built-in constraints
- B: conjunction of CHR and built-in constraints

Constraint Theory $\mathcal{T}$ for Built-In Constraints

Example

$$
\begin{aligned}
X \leq Y &\Leftrightarrow X = Y \mid true \\
X \leq Y \land Y \leq X &\Leftrightarrow X = Y \\
X \leq Y \land Y \leq Z &\Rightarrow X \leq Z
\end{aligned}
$$

$A \leq B \land B \leq C \land C \leq A$
$\longrightarrow$ (transitivity)
$A \leq B \land B \leq C \land C \leq A \land A \leq C$
$\longrightarrow$ (antisymmetry)
$A \leq B \land B \leq C \land A = C$
$\longrightarrow$ (built-in solver)
$A \leq B \land B \leq A \land A = C$
$\longrightarrow$ (antisymmetry)
$A = B \land A = C$

Operational Semantics

Apply rules until exhaustion in any order (fixpoint computation).

Simplify

$$\frac{(H \Leftrightarrow C \mid B)[x/y] \in P \qquad \mathcal{T} \models G_{builtin} \supset \exists x(H = H' \wedge C)}{H' \wedge G \longrightarrow G \wedge H = H' \wedge B}$$

Propagate

$$\frac{(H \Rightarrow C \mid B)[x/y] \in P \qquad \mathcal{T} \models G_{builtin} \supset \exists x(H = H' \wedge C)}{H' \wedge G \longrightarrow H' \wedge G \wedge H = H' \wedge B}$$

Refined operational semantics [Duck04iclp]: Similar to Prolog, CHR constraints evaluated depth-first from left to right and rules applied top-down in program text order.

Operational Properties

- Computation can be interrupted and restarted at any time. Intermediate results approximate final result.

- Monotonicity and Incrementality
  If $G \longrightarrow G'$
  then $G \wedge C \longrightarrow G' \wedge C$

- Termination, Consistency and Confluence can be analyzed:
  - Termination is as usual difficult in general...
  - For terminating programs, confluence is analyzed on critical pairs

Applications

Many CLP solvers have been written in CHR

- $\mathcal{B}$
- $\mathcal{FD}$
- $\mathcal{R}$ (linear)
- unification
- scheduling
- typing inequalities
- ...

Even outside of the CLP community:
Understanding functional dependencies via Constraint Handling Rules, Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey.
[Journal of Functional Programming 2007].

CHRsss

Many extensions:

- probabilistic
- $\mathrm{CHR}^{\vee}$
- soft constraints

Many implementations

- SWI-Prolog
- Haskell
- Java

CHRsss

Many extensions:

- probabilistic
- $CHR^{\vee}$
- soft constraints

Many implementations

- SWI-Prolog
- Haskell
- Java

Many semantics

- refined
- compositional
- classical logic
- linear logic

Part VIII

Programming Project

Part VIII: Programming Project

26 check_dice

27 dice

28 Optimizing

29 Theory

Generalities

Lost of pretty good submissions (haven't looked at everything thoroughly yet).

Almost no use of the existing literature. Could have helped especially for the third part. Programming is like research: do not forget to check the state-of-the-art.

The pure CLP($\mathcal{H}$) part seems to have been easy for everyone.

Almost every possible optimization was tried, but not all in any solution.

Since in many cases the "optimal optimizations" are a question of combination, I will simply give you ideas instead of a specific solution.

check_dice

```
check_dice([H | T]) :-
   check_dice_pairs([H | T], H).


check_dice_pairs([A], B) :-
   check_dice_pair(A, B).

check_dice_pairs([A, B | L], C) :-
   check_dice_pair(A, B),
   check_dice_pairs([B | L], C).


check_dice_pair(A, B) :-
   check_dice_pair2(A, B, Won, Tot),
   P is Won / Tot,
   P > 0.5,
   format(”~w␣>␣~w␣with␣p:␣~w~n”, [A, B, P]).
```

```
check_dice_pair2([], _, 0, 0).
check_dice_pair2([A | LA], LB, W, T) :-
   check_dice_pair3(A, LB, WW, TT),
   check_dice_pair2(LA, LB, WWW, TTT),
   W is WW + WWW,
   T is TT + TTT.

check_dice_pair3(_, [], 0, 0).
check_dice_pair3(A, [B | LB], W, T):-
   check_dice_pair3(A, LB, WW, TT),
   T is TT + 1,
   (
      A > B      % @> is the term ordering
   ->
      W is WW + 1
   ;
      W = WW
   ).
```

dice

```
dice(N, F, D) :-
   M is N*F,
   dice_init(N, F, D, M),
   cycle(D, _W),
   flatten(D, DD),
   all_different(DD),
   break_sym(D),
   labeling([ff], DD),
   check_dice(D).

dice_init(0, _, [], _).

dice_init(N, F, [D | DL], M) :-
   N > 0,
   length(D, F),
   D ins 1..M,
   NN is N-1,
   dice_init(NN, F, DL, M).
```

```
cycle([H | T], W) :-
   cycle2([H | T], H, W).


cycle2([], _, 0).

cycle2([A], B, W) :-
   win(A, B, W).

cycle2([A, B | L], C, W) :-
   win(A, B, WW),
   cycle2([B | L], C, WWW),
   W #=< WW,           % some way to get the min
   W #=< WWW.          % purely with constraints
```

```
win(A, B, C) :-
   win2(A, B, L),
   length(L, N),
   M is N // 2,
   fd_cardinality(L, C),
   C #> M.

win2([], _, []).

win2([A | LA], LB, LC) :-
   win3(A, LB, L),
   append(L, LL, LC),
   win2(LA, LB, LL).

win3(_, [], []).

win3(A, [B | LB], [A #> B | LC]):-
   win3(A, LB, LC).
```

Cardinality

```
fd_cardinality([], 0).

fd_cardinality([H | T], C) :-
   B #<==> H,
   C #= B + D,
   fd_cardinality(T, D).
```

This is very different from creating a choice point like:

```
win3(A, B, C) :-
   A #> B,
   ...
win3(A, B, C) :-
   A #=< B,
   ...
```

```
break_sym([[1 | D] | DD]) :-
   order_rec([[1 | D] | DD]).


order_rec([]).

order_rec([H | T]) :-
   order(H),   % same as chain(H, #=<)
   order_rec(T).


order([]).

order([_]).

order([H1, H2 | L]) :-
   H1 #=< H2,
   order([H2 | L]).
```

dice

```
dice(N, F, D) :-
   M is N*F,
   dice_init(N, F, D, M),
   cycle(D, W),
   flatten(D, DD),
   all_different(DD),          % not permutation
   break_sym(D),
   labeling([ff,max(W)], DD), % B&B not findall
   check_dice(D),
   write(W),
   nl.
```

Not bad (optimizes 4 dice with 5 faces or 3 dice with 6 faces in less than 30s)

Alternative approach

Hand-code the branch and bound procedure on W and search with W instantiated!

Does gain a little in SWI

Alternative approach

Hand-code the branch and bound procedure on W and search with W instantiated!
Does gain a little in SWI
Does gain 2 orders of magnitude in GNU…

```
> gprolog --consult-file dice.pl
...
| ?- dice(3, 6, L).
[1,9,10,11,12,14] > [5,6,7,8,13,18] with p: 0.5833333
[5,6,7,8,13,18] > [2,3,4,15,16,17] with p: 0.5833333
[2,3,4,15,16,17] > [1,9,10,11,12,14] with p: 0.583333
21

L = [[1,9,10,11,12,14],[5,6,7,8,13,18],[2,3,4,15,16,17

(85 ms) yes
```

Optimizing the generation

One can actually create cycles of 3 or 4 dice, whatever the number of faces

They can be chained for any number of dice (except 5)

One can also add faces to a given cycle (carefully)

Or on the contrary limit the number of different faces

- If $S \neq 4$ you need
  - 5 if N mod 3 = 0
  - 6 otherwise
  - 7 values for N=5
- If S=4 then you need one more

These cycles have a low winning ratio (close to 0.5) not useful for optimization.

Optimizing the optimization

Upper bound given by reasoning on median value (see [Trybula 1965, Savage 1994]):

$$p \leq \frac{3}{4} - \frac{1}{2n} - \alpha\frac{1}{4n^2}$$

where $\alpha = 1$ if S is odd and 0 if S is even.

This bound is reached if N is equal to S

Optimizing the optimization

Upper bound given by reasoning on median value (see [Trybula 1965, Savage 1994]):

$$p \leq \frac{3}{4} - \frac{1}{2n} - \alpha\frac{1}{4n^2}$$

where $\alpha = 1$ if S is odd and 0 if S is even.

This bound is reached if N is equal to S or greater

If N $<$ S this limit might be unreachable (e.g. 24 wins for 4 dice with 6 faces, but 21 if only 3 dice…)

Some lower bounds can be obtained through the same kind of reasoning as before, but not perfect (e.g. 14 for 3 dice with 5 faces, but 15 can be obtained and upper bound would be 16)

Other ones come from other systematic constructions

Minizinc

From minizinc.org's tutorial

```
include "alldifferent.mzn";

int: n; % number of dices
int: s; % number of sides

array[1..n,1..s] of var 1..n*s: dices;

constraint dices[1,1] = 1;
constraint alldifferent([ dices[i,j] | i in 1..n, j in 1..s ]);
constraint forall ( i in 1..n, j in 1..s-1 ) ( dices[i,j] < dices[i,j+1] );
constraint forall ( i in 1..n ) (
   sum ([1 | j,k in 1..s where dices[i,j] > dices[i+1 mod n, k]]) > n*s / 2
);

solve satisfy;

output ["dices = ",show(dices),"\n"];
```

Many global constraints in "globals.mzn".

Used for the CP contest each year.

Issues

What happens when a solver does not have a global constraint?

Not very declarative, even though loops and comprehensions have been added.

Search strategies are crucial, yet limited possibilities via *search annotations*.

e.g. `solve :: int_search(q, first_fail, indomain_min, complete) satisfy;`

Hot topics in CP

Global constraint decomposition with same AC propagation (there might still be an overhead, typically linked to the number of variables).

"Search is dead"
Getting rid of search strategies through Lazy Clause Generation (the CP equivalent of nogood learning in SAT).

High-level search strategies definitions through reified constraints. In CLPZinc, a search strategy is a constraint.

CLPZinc

```
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).

dichotomy_list([], _Min, _Max).
dichotomy_list([H | T], Min, Max) :-
   dichotomy(H, Min, Max),
   dichotomy_list(T, Min, Max).

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

Compiles to…

```
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;
```

And/Or tree

```
        indexical_min(X1, x)
        indexical_max(X2, x)
                          x <= (X1 + X2) div 2
                     ∨
                          x > (X1 + X2) div 2
        indexical_min(X3, x)
   ∧    indexical_max(X4, x)
                          x <= (X3 + X4) div 2
                     ∨
                          x > (X3 + X4) div 2
        indexical_min(X5, x)
        indexical_max(X6, x)
                          x <= (X5 + X6) div 2
                     ∨
                          x > (X5 + X6) div 2
```