High-Level Implementation of Consistency Techniques

Joachim Schimpf
Visiting Researcher at NICTA Victoria
Motivation

Propagation behaviour in CP systems:

- Strength/cost of propagation hard-wired
  - only occasionally configurable
- System-specific set of implemented constraints
  - no standard set of global constraints
  - but constraint catalog lists 235 of them [Beldiceanu et al]
- Efficient propagation algorithms are constraint-specific
  - take a long time to develop
- Implemented procedurally or via rules
  - correctness/maintainability

⇒ declarative prototyping facility can be useful
Overview

- Constraint definitions
- Reification and its shortcomings
- GP - Generalised Propagation
- Applications of GP
- GP Algorithm
Constraint definitions

- Basic constraints, e.g. equality, domains (decidable)
  \[ x=5, \quad y::1..9 \]

- Built-in constraints (e.g. bounds/arc consistent)
  \[ X \#>= 3*y+7, \quad \text{alldifferent([X,Y,Z])} \]

- User-defined constraints
  - Extensional definition (table, disjunctive)
    \[
    \%
    \% \text{product(Name,Resource1,Resource2,Profit)}
    \%
    \text{product(101, 3, 7, 36).}
    \]
    
    \[
    \%
    \text{...}
    \%
    \text{product(999, 5, 2, 23).}
    \]

  - Intensional definition (logical combinations of built-in constraints)
    \[
    \text{no_overlap(S1,S2,D)} :-
    \%
    S1+D \#>= S2 ; S2+D \#>= S1.
    \]
Propagation choices

When?
- Once before search (preprocessing)
- Whenever a variable gets instantiated
- Whenever a domain is reduced
- More/Less urgently than other constraints

What?
- Just check current assignment
- Derive further instantiations
- Derive domain reductions

Where?
- Per constraint
- Sub-problem
- Whole problem
Well known: User-defined Constraints via “reification”

The system must provide “reified” versions of constraints, e.g.

\[ =<(X, Y, B) \iff X =< Y \text{ iff } B=1 \]
\[ =<(X, Y, B) \iff X > Y \text{ iff } B=0 \]

The Boolean represents the truth value of the constraint. Similar to Big-M constraints in MIP.

Reified primitives can be connected by combining Booleans:

\[ =<(X+7, Y, B1) , =<(Y+7, X, B2) , B1+B2 \geq 1 \]

The Boolean can be hidden under syntactic sugar:

\[ X+7 \geq Y \text{ or } Y+7 \geq X. \]
Merits/Limits of reified combinators

Clever example – a lexicographic ordering constraint for vectors:

\[
\text{lex\_le}(Xs, Ys) :-
( \text{foreach}(X, Xs), \text{foreach}(Y, Ys), \text{fromto}(1, Bi, Bj, 1) \text{ do}
  Bi \neq (X \#< Y + Bj)
).
\]

But often poor propagation behaviour:

?- [X,Y]::1..10, ( X + 7 \#=< Y or Y + 7 \#=< X ).
X = X{1 .. 10}
Y = Y{1 .. 10}
There are 3 delayed goals.

Propagation happens only when Boolean gets instantiated, i.e. when one of the primitives is entailed/disentailed.
Generalised Propagation (GP)

A generic algorithm to extract information from disjunctive specifications

\[
\begin{align*}
    c(1,2) . \\
    c(1,3) .
    c(3,4) .
\end{align*}
\]

?- c(X,Y) infers fd.
X = X{[1, 3]}
Y = Y{2 .. 4}

First described in [LeProvost&Wallace 93]
Implemented in ECLiPSe system as library “propia”
The *infers* Annotation

GP annotation says *what* you want to infer:

- **Goal infers Language**
  
  Use strongest available representation in Language (e.g. fd for finite domains) covering all solutions

Weaker (and cheaper) variants are available

- **Goal infers consistent**
  
  Fail as soon as inconsistency can be proven

- **Goal infers unique**
  
  Instantiate as soon as unique solution exists
Most Specific Generalisation over Finite Domains and Structured Terms

```
3 1 2
X{[1,2]}

X{1..3}  "most specific generalisations"
```

```
X(1,3)   X(3,4)
X(1,3), Y(2,4)  
```


Arc consistency (i)

- Arc consistency from extensional definition:

```prolog
% extensional constraint spec
c(1,2).
\(c(1,3).\)
c(3,4).

% arc-consistent version
ac_c(X,Y) :- c(X,Y) infers fd.
```

?- ac_c(X, Y).
X = X{[1, 3]}
Y = Y{[2..4]}
There is 1 delayed goal.

?- ac_c(X, Y), X = 1.
X = 1
Y = Y{[2,3]}
There is 1 delayed goal.
Constructive disjunction with GP

The inferences from disjunctive branches are merged constructively:

?- [A,B]::1..10, (A + 7 #=< B ; B + 7 #=< A) infers fd.

   1..3     8..10     1..3     8..10

A = A{[1 .. 3, 8 .. 10]}
B = B{[1 .. 3, 8 .. 10]}

Note difference with reification – no inference because neither side is (dis)entailed:

?- [A,B]::1..10, (A + 7 #=< B or B + 7 #=< A).

A = A{1 .. 10}
B = B{1 .. 10}
Arc-consistency (ii)

Arc consistency on top of weaker consistency (e.g. test, forward checking)

```prolog
ac_constr(Xs) :-
    (    weak_constr(Xs),
       labeling(Xs)
    ) infers fd.
```

Or, usually more efficient:

```prolog
ac_constr(Xs) :-
    (    weak_constr(Xs),
       member(X, Xs),
       indomain(X),
       once labeling(Xs)
    ) infers fd.
```
Singleton Arc-consistency

Singleton arc consistency from arc consistency, on a subproblem:

```prolog
sac_constr(Xs) :-
    ( ac_constr(<some Xs>), ..., ac_constr(<some Xs>),
      member(X, Xs),
      indomain(X)
    ) infers ic.
```

If performed on the whole problem, simpler variant: *shaving*

```prolog
shave(Xs) :-
    ( foreach(X,Xs) do
        findall(X, indomain(X), Values),
        X :: Values
    ).
```

Shaving often effective as a preprocessing step before actual search. E.g. sudoku solvable with ac-alldifferent and shaving – no deep search needed [Simonis].
Arc-consistency from arc-consistency

Combining constraints to form a sub-problem.
Make result arc-consistent again:

E.g. a constraint for sudoku:

\[
\text{overlapping\_alldifferent}(Xs, Ys) :- \\
\quad \text{intersect}(Xs, Ys, Overlaps), \\
\quad \left( \\
\quad \quad \text{alldifferent}(Xs), \text{alldifferent}(Ys), \\
\quad \quad \text{labeling}(Overlaps) \\
\quad \right) \text{ infers ic.}
\]
GP Applications Summary

- Disjunctive combinations
  - extensional or intensional constraint definitions

- Factoring subproblems
  - effectively create problem-specific global constraints
  - approximate their solution set repeatedly
  - export that approximation repeatedly to the full problem

- Conjunctive combination of overlapping constraints
  - to form larger global constraints

- Prototyping AC constraints
  - expensive when done naively, need good generic GP algorithm
  - better to use a less disjunctive specification, see below
Graph/automaton method (i)

- Beldiceanu et al, 2004: Deriving Filtering Algorithms from Constraint Checkers

```prolog
global_contiguity(Xs) :-
    StateEnd :: 0..2,
    (fromto(Xs, [X|Xs1], Xs1, []),
     fromto(0, StateIn, StateOut, StateEnd)
    do
    (StateIn = 0, (X = 0, StateOut = 0 ; X = 1, StateOut = 1 )
    ; StateIn = 1, (X = 0, StateOut = 2 ; X = 1, StateOut = 1 )
    ; StateIn = 2, X = 0, StateOut = 2
    ) infers ac
    ).
```
inflexion(N, Xs) :-
    StateEnd :: 0..2,
    ( fromto(Xs, [X1,X2|Xs1], [X2|Xs1], [_.]),
      foreach(Ninc, Nincs),
      fromto(0, StateIn, StateOut, StateEnd)
    do
      (X1 #< X2) #= (Sig #= 1),
      (X1 == X2) #= (Sig #= 2),
      (X1 #> X2) #= (Sig #= 3),
      (StateIn = 0,
       (Sig=1, Ninc=0, StateOut=1
        ; Sig=2, Ninc=0, StateOut=0
        ; Sig=3, Ninc=0, StateOut=2 )
       ; StateIn = 1,
       (Sig=1, Ninc=0, StateOut=1
        ; Sig=2, Ninc=0, StateOut=1
        ; Sig=3, Ninc=1, StateOut=2 )
      ; StateIn = 2,
       (Sig=1, Ninc=1, StateOut=1
        ; Sig=2, Ninc=0, StateOut=2
        ; Sig=3, Ninc=0, StateOut=2 )
       ) infers ac
    ),
    N #=> sum(Nincs).
Naïve GP Algorithm

Goal infers Language

Find all solutions to Goal, and put them in a set
Find the most specific generalisation of all the terms in the set

E.g.  \textit{member}(X,[1,2,3]) \textit{infers} fd

Find all solutions to \textit{member}(X,[1,2,3]): \{1,2,3\}
Find the most specific generalisation of \{1,2,3\}: \textit{X} \{[1,2,3]\}

Efficient when all solutions can be tabled.
Robust GP Algorithm: Topological B&B

Goal infers Language

Find one solution $S$ to Goal
The current most specific generalisation $MSG = S$
Repeat
    Find a solution $NewS$ to Goal
    which is NOT an instance of $MSG$
    Find the most specific generalisation $NewMSG$
    of $MSG$ and $NewS$
    $MSG := NewMSG$
until no such solution remains
Resources

- Functionality available in the ECLiPSe system
  Main web site [www.eclipse-clp.org](http://www.eclipse-clp.org)
  Tutorial, papers, manuals, mailing lists
  Sources at [www.sourceforge.net/eclipse-clp](http://www.sourceforge.net/eclipse-clp)

- ECLiPSe is open-source (MPL) and freely usable
  Owned/sponsored by Cisco Systems

- References