FOUNDATIONS OF CONSTRAINT PROGRAMMING AND CONSTRAINT LOGIC PROGRAMMING

kvenable@math.unipd.it

K. Brent Venable University of Padova, Italy

General Outline

- Foundations of Constraint Programming
  - what is constraint programming
  - short history
  - search
  - inference
  - combining search and inference
- Foundations of Constraint Logic programming
  - CP+LP=CLP
  - short history
  - operational semantics
  - semantics of success
  - semantics of finite failure

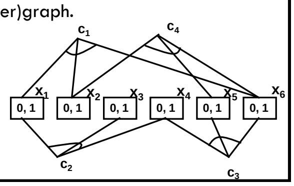
CP: CONSTRAINT PROGRAMMING

What is a constraint?

- **Constraint is an arbitrary relation over a set of variables.**
  - **domain of a variable**: set of possible values it can take
  - the constraint restricts the possible combinations of values
- **Examples:**
  - X is less than Y
  - a sum of angles in the triangle is 180°
  - the temperature in the warehouse must be in the range 0-5°C
  - John can attend the lecture on Wednesday after 14:00
- Constraint can be described:
  - **intentionally** (as a mathematical/logical formula), e.g., X<Y
  - **extensionally** (as a table describing compatible tuples)
  - Example : D(X)=D(Y)={1,2}, constraint "X less than Y", {(X=1,Y=2)}

Constraint Satisfaction Problem

- CSP (***Constraint Satisfaction Problem***) *consists of:*
  - a finite set of variables
  - domains - a finite set of values for each variable
  - a finite set of constraints
- *A **solution** to* CSP is a complete assignment of variables satisfying all the constraints.
- CSP is often represented as a (hyper)graph.
- *Example:*
  - variables $x_1,\ldots,x_6$ domain {0,1}
    Constraints :
  - Constraints: c_1: $x_1+x_2+x_6=1$,
    c_2: $x_1-x_3+x_4=1$, c_3: $x_4+x_5-x_6>0$
    c_4: $x_2+x_5-x_6=0$



Example of CSP: cryptoarithmetic problem

SEND + MORE = MONEY

assign different single-digit positive integers to different letters
S and M are not zero

This problem can be modelled by the following CSP

Variables `E,N,D,O,R,Y,S,M,P1,P2,P3`

Domains

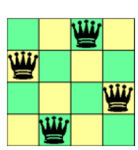
```
D(E)=D(N)=D(D)=D(O)=D(R)=D(Y)={0,...,9}
D(S)=D(M)= {1,...,9},
D(P1)=D(P2)=D(P3)={0,1}
```

Constraints `all_different(S,E,N,D,M,O,R,Y)`

```
      D+E = 10*P1+Y
   P1+N+R = 10*P2+E
   P2+E+O = 10*P3+N
   P3+S+M = 10*M +O
```

Example of CSP: n Queens Problem

- Place n queens in an nxn chessboard such that they do not attack each other
- Variables: x1,...,xn (one per column)
- Domains: [1..n] (row position of a queen)
- Constraints:
  - xi ≠ xj for all i,j (no attack on a row)
  - xi-xj ≠ i-j (no attack on the SW-NE diagonal)
  - xi-xj ≠ j-i (no attack on a NW-SE diagonal)



The early days of CP(1)

- The very early days: Theseus used backtrack to find his way in the labyrinth in Crete
- 1848: chess player Bazzel proposed the 8-queens problems
- 1963 Sutherland's Ph.D. thesis "SketchPad: a man-machine graphical communication system"
- Two main streams of research:
  - **The language stream:**
    - 1970: Fikes proposes the REF-ARF language (1st issue fo AIJ!) REF language part of a general problem solving system using constraint satisfaction and propagation
    - Kowalski: constraints for theorem proving
    - Sussman and Steel: the CONSTRAINTS language
    - Borning: extends Smalltalk to ThingLab using constraints

The early days of CP(2)

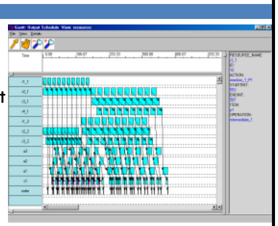
- The algorithm stream
  - 1975: Waltz proposes arc consistency in his PH.D. thesis on scene labeling
  - Montanari: "Networks of constraints: fundamental properties and applications to picture processing"
    - path consistency
    - general framework for constraints
  - Mackworth: "Consistency in networks of relations"
    - a new algorithm for arc consistency
  - Freuder: generalizes arc and path consistency to k-consistency
  - Rosenfeld, Hummel and Zucker: introduce soft constraint as different levels of compatibility

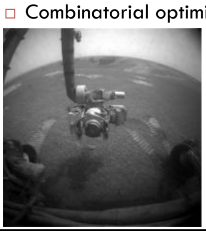
Why should you care about constraint programming?

- Sooner or later you will be asked to solve some horribly complicated problem…
- CP provides a very general for modeling problems
  - CP may help you understand the problem you have to solve
- Many powerful solving techniques have been developed for problems modeled via CP
  - A CP solver may actually solve the problem for you
- This is why CP has proven useful in many application domains

Constraints in A.I. planning and scheduling

- *Scheduling problem* = a set of activities has to be processed by a limited number of resources in a limited amount of time.
- Combinatorial optimisation





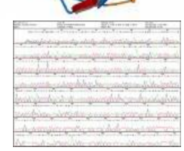
***Planning problem* =**
find a set of activities to achieve a given goal

MER project
- **CP helped Spirit and Opportunity figure when it was better to do things**

Constraints in bioinformatics

- *Design of a 3D protein structure* from the sequence of amino-acids (3D structure determines features of proteins)

- Analysing a sequence of DNA, estimating a distance between DNAs, comparing DNAs



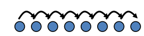
Search

- Basic strategy
  - assign values to variables: enumerate solutions
  - see what happens: use constraints as tests
- *Local search*
  - explore the search space by small steps
- *systematic search*
  - explores the space of all assignments systematically
- *non-systematic search*
  - some assignments may be skipped during search

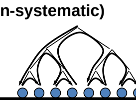
Systematic search

- Explore systematically the space of all assignments
- systematic = every valuation will be explored sometime
- *Features:*
  - + complete (if there is a solution, the method finds it)
  - - it could take a lot of time to find the solution
- *Basic classification:*

Explore complete assignments
generate and test
same search space is used by local search (non-systematic)



Extending partial assignments
tree search



Generate and test (GT)

- The most general problem solving method
  - 1) generate a candidate for solution
  - 2) test if the candidate is really a solution
- How to apply GT to CSP?
  - 1) assign values to all variables
  - 2) test whether all the constraints are satisfied
- GT explores complete but inconsistent assignments until a (complete) consistent assignment is found.

Pros and Cons GT

- *The greatest weakness of GT is exploring too many "visibly" wrong assignments.*
- *Example:*
  X,Y,Z::{1,2}   X = Y, X ≠ Z, Y > Z

X	1	1	1	1	2	2	2
Y	1	1	2	2	1	1	2
Z	1	2	1	2	1	2	1

How to improve generate and test?

smart generator

smart (perhaps non-systematic) generator that uses result of test
↳ local search techniques

earlier detection of clash

constraints are tested as soon as the involved variables are instantiated → backtracking-based search

Local search techniques

Local search

- *One way to overcome GT cons*
- Assume an assignment is inconsistent
- The next assignment can be constructed in such a way that constraint violation is smaller.
  - only "small" changes (local steps)of the assignment are allowed
  - next assignment should be "better" than previous
    - better = more constraints are satisfied
  - assignments are not necessarily generated systematically
    - we lose completeness but we (hopefully) get better efficiency

Local search terminology

- **Search Space S:** set of all complete variable assignments
- **Set of solutions Sol:**
  - subset of the search space
  - all assignments satisfying all the constraints
- **Neighborhood relation:** a subset of SxS indicating what assignments can be reached by a search step given the current assignment during the search procedure
- **Evaluation function:** mapping each assignment to a real number representing "how far the assignment is from being a solution"
- **Initialization function:** which returns an initial position given a possibility distribution over the assignments
- **Step function:** given an assignment, it neighborhood and the evaluation function returns the new assignment to be explored by the search
- **Set of memory states (optional):** holding information about the state of the search mechanism.
- **Termination criterion:** stopping the search when satisfied

Local search for CSPs

- **Neighborhood of an assignment:** all assignments differing on one the value of one variable (1-exchange-neighborhood)
- **Evaluation function:** mapping each assignment to the number of constraints it violates
- **Initialization function:** returns an initial assignment chosen randomly
- **termination criterion:** if a solution is found or if a given number of search steps is exceeded.
- **The different algorithms are characterized by the step function and use of memory.**

Hill Climbing

- The basic technique of local search.
  - starts at a randomly generated assignment
  - At each state of the search
    - **Iterative Best-improvement**: move to the assignment in the neighbourhood violating the minimum number of constraint
    - **Iterative-First-improvement:** choose the first improving neighbour in a given order
    - if multiple choices choose one randomly

- *neighbourhood* = differs in the value of any variable
- neighbourhood size = $\Sigma_{i=1..n}(D_i-1)$ (= n*(d-1))

Min-Conflicts (Minton, Johnston, Laird 1997)

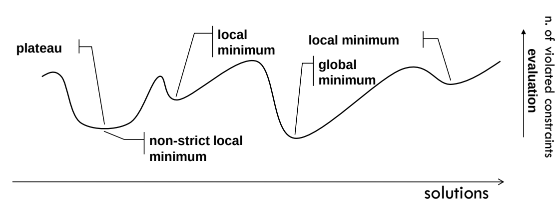
- **Conflict set of an assignment**: set of variables involved in some constraint violating that assignment
- Min-conflict LS procedure:
  - starts at randomly generated assignment
  - at each state of the search
  - selects a variable from the current conflict set
  - selects a value for that variable that minimizes the number of violated constraints
  - if multiple choices choose one randomly

- *neighbourhood* = different values for the selected variable
- neighbourhood size = (d-1)

Local minima

- The evaluation function can have:
- local minimum - a state that is not minimal and there is no state with better evaluation in its neighbourhood
- strict local minimum - a state that is not minimal and there are only states with worse evaluation in its neighbourhood
- global minimum - the state with the best evaluation
- plateau - a set of neighbouring states with the same evaluation

Graphically…



Escaping local minima

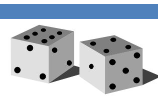
- A local search procedure may get stuck in a local minima
- Techniques for preventing stagnation
  - restart
  - allowing non improving steps → random walk
  - changing the neighborhood → tabu search
  - changing the evaluation function →penalty-based search strategies

Restart

- Re-initialize the search when the after MaxSteps (non-strictly improving) steps
- New assignment chosen randomly
- Can be combined both with hill-climbing and Min-conflicts
- It is effective if MaxSteps is chose correctly and often it depends on the instance

Random walk

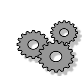
- Add some “noise” to the algorithm!



- *Random walk*
  - a new assignment from the neighbourhood is selected randomly (e.g., the value is chosen randomly)
  - such technique can hardly find a solution
  - so it needs some guide
- Random walk can be combined with the heuristic guiding the search via probability distribution:
  - *p*: probability of using the random walk (noise setting)
  - *(1-p)* : probability of using the heuristic guide
  - Steepest descent random walk: RW+Hill climbing
  - Min-conflicts random walk

Tabu search

- Being trapped in local minimum can be seen as cycling.
- How to avoid cycles in general?
  - Remember already visited states and do not visit them again.
    - memory consuming (too many states)
  - It is possible to remember just a few last states.
    - prevents „short“ cycles
- Tabu list = a list of forbidden states
  - ⟨variable, value⟩ - describes the change of the state (a previous value)
  - tabu list has a fix length *k* (tabu tenure)
    - „old“ states are removed from the list when a new state is added
  - state included in the tabu list is forbidden (it is tabu)
- Aspiration criterion = enabling states that are tabu
  - i.e., it is possible to visit the state even if the state is tabu
  - *example*: the state is better than any state visited so far (the incumbent candidate solution)



Algorithm TS-GH
Galinier anf Hao 1997

Penalty-based algorithms

- Modify the evaluation function when the search is about to stagnate
- Evaluation of an assignment depends on the constraints
- Associate weights to constraints and change them during the search
- Result: the search “learns” to distinguish important constraints

GENET

- Neural Network
- node→ variable assignment
- CSP variable → cluster of NN nodes corresponding to its assignments
- links→ between assignments violating some constraint
- penalty weights associated to links
- 1 at the beginning
- Assignment → only the nodes corresponding to the assignments are switched on
- Each node receives a signal from the neighboring nodes that are switched on with strength equal to the weight of the link
- For each cluster the nodes with the smallest incoming signal are switched on
- When the search stabilizes in a state, the weights of the links among the active nodes is increased by one
- Solution when the minimum signal is 0 for all clusters

Breakout Method

- Similar to GENET
- Weights are associated to constraints
- Evaluation of an assignment = weighted sum of the violated constraints
- When a local minimum is reached the weights of the violated constraints is increased by one

Localizer (Michel, Van Hentenryck 1997)

- The local search algorithms have a similar structure that can be encoded in the common skeleton. This skeleton is filled by procedures implementing a particular technique.

```
procedure local-search(Max_Tries,Max_Moves)
    s ← random assignment of variables
    for i:=1 to Max_Tries while Gcondition do
        for j:=1 to Max_Moves while Lcondition do
            if eval(s)=0 then
                    return s
            end if
            select n in neighbourhood(s)
            if acceptable(n) then
                    s ← n
            end if
        end for
        s ← restartState(s)
    end for
    return best s
end local-search
```

Systematic search techniques

Backtracking

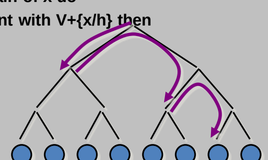
- Key idea: extend a partial consistent assignment until a complete consistent assignment is found
- The most widely used systematic search algorithm
- Basically : depth-first search

- Backtracking for CSP
  - 1) assign values to variables incrementally
  - 2) after each assignment test the constraints over the assigned variables (and backtrack upon failure)
- *Parameters:*
  - In which order to assign variables
  - what is the order of values?
    - problem dependent

Algorithm chronological backtracking

```
procedure BT(X:variables, V:assignment, C:constraints)
    if X={} then return V
    x ← select a not-yet assigned variable from X
    for each value h from the domain of x do
       if constraints C are consistent with V+{x/h} then
           R ← BT(X-x, V+{x/h}, C)
           if R≠fail then return R
    end for
    return fail

call BT(X, {}, C)
```



Backtracking is always better than generate and test!

Cons of backtracking(1)

- thrashing
  - throws away the reason of the conflict
  - *Example:* A,B,C,D,E:: 1..10, A>E
    - BT tries all the assignments for B,C,D before finding that A≠1
  - *Solution:* backjumping (jump to the source of the failure)
- redundant work
  - unnecessary constraint checks are repeated
  - *Example:* A,B,C,D,E:: 1..10, B+8<D, C=5*E
    - when labelling C,E the values 1,..,9 are repeatedly checked for D
  - *Solution:* remember (no-)good assignments

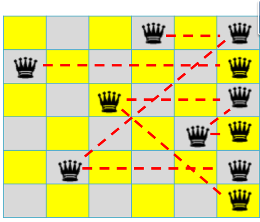
Cons of backtracking(2)

- late detection of the conflict
  - constraint violation is discovered only when the values are known
  - *Example:* A,B,C,D,E::1..10, A=3*E
    - the fact that A>2 is discovered when labelling E
  - *Solution:* forward checking (forward check of constraints)

No-good

- Informally, a No-good is a set of assignments that is not consistent with any solution
- Let $p=\{X_1=a_1, X_2=a_2,\ldots,X_k=a_k\}$ be a deadend of the search tree
- A **jumpback no-good** for p is defined recursively
  - If **p is a leaf node** and C is a constraint violated by p
    - $J(p)=\{X_h=a_h \mid X_h \text{ is in vars(C)}\}$
  - otherwise, le $\{X_{k+1}=v_1, \ldots X_{k+1}=v_j\}$ be all the possible extensions to X_{k+1} tempted by the search
  - $J(p)= \cup_{i=1..j} (J(p\cup \{X_{k+1}=v_i\}) - \{X_{k+1}=v_i\})$

Example of No-good



p={X1=2,X2=5,X3=3,X4=1,X5=4}

$J(p)= \cup_{i=1..j} (J(p\cup \{X_{k+1}=v_i\})- \{X_{k+1}=v_i\})$

$J(p)=(J(p \cup\{X_6=1\}) - \{X_6=1\}) \cup \ldots \cup (J(p\cup \{X_6=6\})-\{X_6=6\})=\ldots$

Choosing constraints in lex order

$J(p)=\{X_h=a_h \mid X_h \text{ is in vars(C)}\}$

$\ldots=\{X_2=5\} \cup \ldots \cup\{X_3=3\}= \{X_1=2,X_2=5,X_3=3,X_5=4\}$

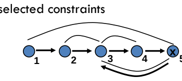
Backjumping (Gaschnig 1979)

- Backjumping is used to remove thrashing.

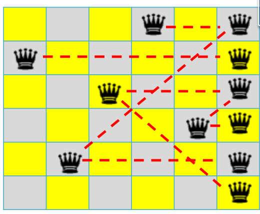
1. identify the source of the conflict (impossible to assign a value)
2. jump to the past variable in conflict

- irrelevant assignments are skipped and undone!

- *Where to jump to when at dead-end p:*
  - Without No-goods
    - select the constraints containing just the currently assigned variable and the past variables
    - select the closest variable participating in the selected constraints
  - With No-goods
    - select the X_i where i is the largest index in J(p)



Example of Backjump with no good



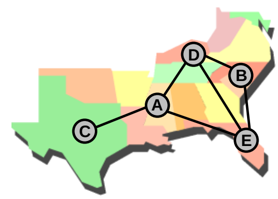
p={X1=2,X2=5,X3=3,X4=1,X5=4}

J(p)={X_1=2,X_2=5,X_3=3,X_5=4}

Undo X_5=4

Weakness of backjumping

- When jumping back the in-between assignment is lost!
- *Example:*
  - colour the graph in such a way that the connected vertices have different colours

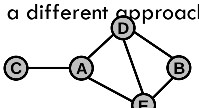


node	vertex		
A	1		1
B	2		1
C	1 2		1 2
D	1 2 3	backjump	1 2
E	1 2 3		1 2 3

During the second attempt to label C superfluous work is done - it is enough to leave there the original value 2, the change of B does not influence C.

Dynamic backtracking

- The same graph (A,B,C,D,E), the same colours (1,2,3) but a different approach.



Backjumping
+ remember the source of the conflict
+ carry the source of the conflict
+ change the order of variables

= DYNAMIC BACKTRACKING

node	1	2	3
A	•		
B		•	
C	A	•	
D	A	B	•
E	A	B	D

jump back
+ carry the conflict source

node	1	2	3
A	•		
B		•	
C	A	•	
D	A	B	AB
E	A	B	

jump back
+ carry the conflict source
+ change the order of B, C

node	1	2	3
A	•		
C	A	•	
B	•	A	
D	A	•	
E	A	B	•

• selected color
AB a source of the conflict

The vertex C (and the possible sub-graph connected to C) is not re-coloured.

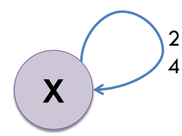
Inference

Constraint propagation

- Transform a CSP into an equivalent simpler CSP
- Main idea: remove elements from domains or tuples from constraints if they cannot participate in any solution
- Aim: to obtain a local consistency property
- *Example:*
  - A in 3..7, B in 1..5 — the variables' domains
  - A<B — the constraint
  - many inconsistent values can be removed
  - we get A in 3..4, B in 4..5
  - *Note:* it does not mean that all the remaining combinations of the values are consistent (for example A=4, B=4 is not consistent)

Node consistency (NC)

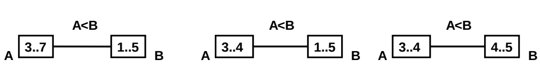
- *Node consistency:*
  - *The vertex* representing the variable X **is *node consistent*** iff every value in the variable's domain D_x satisfies all the unary constraints imposed on the variable X.
  - *CSP* is *node consistent* iff all the vertices are node consistent.



$D(X)=\{\cancel{1},2,\cancel{3},4\}$

Arc consistency (AC)

- *A value v ∈D(X) is said to **have support** in constraint c consistent if there is an assignment satisfying c in which X=v*
- *A constraint c* is ***arc consistent*** iff every value in the domain of each of its variables has support in c
- *CSP* is *arc consistent* iff every constraint is arc consistent.
  - *Usually we say Arc Consistency (AC) for binary constraints and Generalized Arc Consistency if there are non binary constraints*



no arc is consistent — no arc is consistent — arc is consistent

Arc Revision

- How to make the domain of a variables arc consistent w.r.t. a constraint?
- Delete all the values *x* from the domain D that are inconsistent with all the assignment to the other variables.
- Binary case:
  - delete v from D(X) if there is no value *w* in D(Y) such that the valuation X = v, Y = w satisfies the binary constrains on X and Y
- **Arc (X,c)**
- Revise(X,c): removes from D(X) all the values without support in c
  - Returns
    - true if the domain has been reduced
    - false otherwise

AC-1

- Loop over all arc revisions (pairs (variable, constraint)) until no change occurs.

What is wrong with AC-1?

- If a single domain is pruned then revisions of all the arcs are repeated even if the pruned domain does not influence most of these arcs.
- *What arcs should be reconsidered for revisions?*
- The arcs involving variables whose consistency is affected by the domain pruning
- i.e., the arcs with variables involved in some constraints with the reduced variable.

AC-2 and AC-3

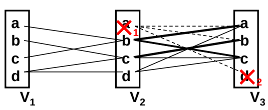
- AC-2(Mackworth '77)
  - In every step, the arcs involving a given variable are processed (i.e. a sub-graph of visited nodes is AC)
- AC-3 (Mackworth '77)
  1. Put all arcs in a queue Q
  2. While Q not empty
  3. (X,c)=Pop(Q)
  4. If Revise((X,c)) wipes out the domain of X: stop
  5. else
  6. if revise(X,c) returns true add to Q all arcs (Y,c') such that c' involves X and Y

Complexity of AC-3

- For binary constraint networks
- Time: $O(ed^3)$
- e: number of constraints
- d: domain size
- Proof:
  - (X,c) is revised only when it is in the Q
  - (X,c) is inserted in the Q only when the domain of some Y involved with X in c has been revised
  - This can happen at most *d* times
  - there are *2e* arcs (X,c)
  - Thus *2ed* revisions each costing at most d^2
- Space: O(e) : the queue contains at most e elements

Looking for (and remembering of) the support

With AC-3 many pairs of values are tested for consistency in every arc revision and these tests are repeated every time the arc is revised.



1. When the arc (V_2,c_{12}) is revised, the value *a* is removed from domain of V_2.

2. Now the domain of V_3, should be explored to find out if any value *a,b,c,d* loses the support in V_2.

Observation:

The values *a,b,c* need not be checked again because they still have a support in V_2 different from *a*.

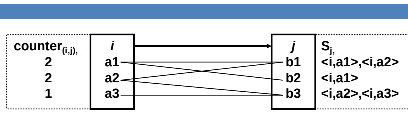
The support set **for $a \in D_i$ is the set $\{<x_j,b> \mid b \in D_j, (a,b) \in C_{i,j}\}$**

Compute the support sets once and then use them during re-revisions.

Support sets

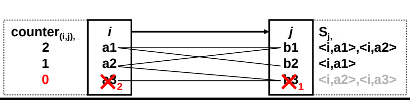
- For each constraint c on variables X and Y, for each value of v in D(X) (and D(Y))
- Compute:
  - **Counter(X,v,Y):** how many supports does v have in c
  - **Support set (or list) S(X,v,Y):** set of values of Y supported by v in c
- if the v disappears then these values lose one support

AC-4 (Mohr and Anderson '86)



Using the support sets:

1. **Assume b3 is deleted from the domain of j (for some reason).**
2. **Look at $S_{j,b3}$ to find out the values that were supported by b3 (i.e. <i,a2>,<i,a3>).**
3. **Decrease the counter for these values (they lost one support).**
4. **If any counter is zero (a3) then delete the value and repeat the procedure for the values it supported (i.e., go to 1).**



Complexity of AC-4

- On binary constraint networks
- **Time: O(ed^2)**
- e: number of constraints
- d: domain size
  - for each value for each constraint I must look for support only once: at most *e2d* times
  - Looking for support takes d
  - Thus O(ed^2)
    - optimal!
- **Space: O(ed^2)**
  - Maximal total size of the support lists

Other arc consistency algorithms

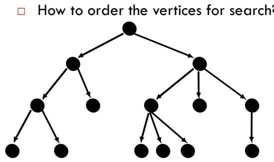
- *AC-4: optimal worst case but bad average case and bad space complexity*
- *AC-6 (Bessiere 1994)*
  - improves memory complexity and average time complexity of AC-4
  - keeps one support only, the next support is looked for when the current support is lost
  - Complexity
    - time $O(ed^2)$
    - Space $O(ed)$
- *AC-2001*
  - Similar to AC-3
  - Pointer Last[X,v,Y]: is the "smallest" value of Y supporting v in c
  - Complexity as AC-6

Directional arc consistency (DAC)

- *Observation 1:* arc revisions have a directional character but CSP is not directional.
- *Observation 2:* AC has to repeat arc revisions; the total number of revisions depends on the number of arcs but also on the size of domains.
- Weakening AC assuming an order over the variables
- *Definition*: A binary CSP is ***directional arc consistent*** using a given order of variables iff for every constraint c(Xi,Xj) such that Xi<Xj the (Xi,c) is arc consistent in c.

How to use DAC

- AC is stronger than DAC (if CSP is AC then it is DAC as well)
- So, is DAC useful?
  - DAC-1 is surely much faster than any AC-x
  - there exist problems where DAC is enough
- *Example:* If the constraint graph forms **a tree** then DAC is enough to solve the problem without backtracks.
- How to order the vertices for DAC?
- How to order the vertices for search?



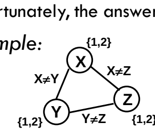
1. Apply DAC in the order from the leaf nodes to the root.

2. Label vertices starting from the root.

DAC guarantees that there is a value for the child node compatible with all the parents.

Is arc consistency enough?

- By using AC we can remove many incompatible values
  - Do we get a solution?
  - Do we know if there exists a solution?
- Unfortunately, the answer to both above questions is NO!
- *Example:*



CSP is arc consistent but there is no solution

So what is the benefit of AC?

Sometimes we have a solution after AC

- **a domain is empty → no solution exists**
- **all the domains are singleton → we have a solution**

In general, AC prunes the search space → equivalent easier problem

Singleton Arc Consistency

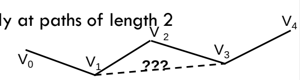
- Another possible relaxation of AC
- A CSP P is SAC iff for every variable X and for every value v in D(X) then $P_{|X=v}$ is not arc inconsistent

Consistency techniques in practice

- N-ary constraints are processed directly!
  - The constraint C_Y is arc consistent iff for every variable *i* constrained by C_Y and for every value $v \in D_i$ there is an assignment of the remaining variables in C_Y such that the constraint is satisfied.
  - *Example:* A+B=C, A in 1..3, B in 2..4, C in 3..7 is AC
- Constraint semantics is used!
  - Interval consistency
    - working with intervals rather than with individual values
    - interval arithmetic
    - *Example:* after change of A we compute A+B → C, C-A → B
  - bounded consistency
    - only lower and upper bound of the domain are propagated
  - Such techniques do not provide full arc consistency!
- It is possible to use different levels of consistency for different constraints!

Path consistency (PC)

- How to strengthen the consistency level?
- Require consistency over more than one constraint
- ***Path*** $(V_0,V_1,\dots,V_m)$ is ***path consistent*** iff for every pair of values $x \in D_0$ a $y \in D_m$ satisfying all the binary constraints on V_0,V_m there exists an assignment of variables $V_1,\dots,V_{m-1}$ such that all the binary constraints between the neighbouring variables V_i,V_{i+1} are satisfied.
- CSP is ***path consistent*** iff every path is consistent.
- Path consistency does not guarantee that all the constraints among the variables on the path are satisfied; only the constraints between the neighbouring variables must be satisfied.
- For PC it is sufficient to look only at paths of length 2



Montanari

Operations over the constraints

Intersection Rij & R'ij

bitwise AND

A<B	&	A≥B-1	→ B-1≤A<B
011		110	010
001	&	111 =	001
000		111	000

Composition $R_{ik} * R_{kj} \to R_{ij}$

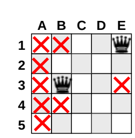
binary matrix multiplication

A<B	*	*B<C →*	*A<C-1*
011		011	001
001	*	001 =	000
000		000	000

The induced constraint is joined with the original constraint

R_{ij} & $(R_{ik} * R_{kj}) \to R_{ij}$

R_{25}	&	(R_{21}	* R_{15})	→	R_{25}
01101		00111	01110		01101
10110		00011	10111		10110
11011	&	10001	* 11011	=	01010
01101		11000	11101		01101
10110		11100	01110		10110



$R_{ij} = R^T_{ji}$, R_{ii} is a diagonal matrix representing the domain

REVISE((i,j)) from AC is equivalent to $R_{ii} \leftarrow R_{ii}$ & $(R_{ij} * R_{jj} * R_{ji})$

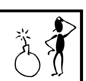
PC-1 and PC-2

- **PC-1** (Mackworth 77)
  - How to make the path (i,k,j) consistent?
    - $R_{ij} \leftarrow R_{ij}$ & $(R_{ik} * R_{kk} * R_{kj})$
  - How to make a CSP path consistent?
    - Repeated revisions of all paths (of length 2) while any domain changes.
- **PC-2** (Mackworth 77)
  - Paths in one direction only (attention, this is not DPC!)
  - After every revision, the affected paths are re-revised

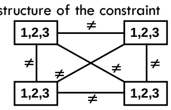
Other path consistency algorithms

- ***PC-3*** *(Mohr, Henderson 1986) and* ***PC-4*** *(Han, Lee 1988)*
  - based on computing supports for a value (like AC-4)
- ***PC-5*** *(Singh 1995)*
  - uses the ideas behind AC-6
  - only one support is kept and a new support is looked for when the current support is lost

Drawbacks of path consistency



- *Memory consumption*
  - because PC eliminates pairs of values, we need to keep all the compatible pairs extensionally, e.g. using {0,1}-matrix
- *Bad ratio strength/efficiency*
  - PC removes more (or same) inconsistencies than AC, but the strength/efficiency ratio is much worse than for AC
- *Modifies the constraint network*
  - PC adds redundant arcs (constraints) and thus it changes connectivity of the constraint network
  - this complicates using heuristics derived from the structure of the constraint network (like tightness, graph width etc.)
- *PC is still not a complete technique*
  - A,B,C,D in {1,2,3}
    A≠B, A≠C, A≠D, B≠C, B≠D, C≠D
    is PC but has not solution

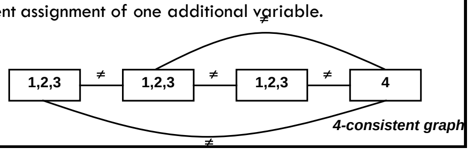


Restricted path consistency(Berlandier '95)

- A binary CSP is Restricted Path Consistent iff
  - it is arc consistent
  - for every constraints c(XY)
    - for each v in D(X) which has a unique support w in D(Y)
    - for each variable Z connected to X and Y
    - there is a value z of D(z) such that (v,z) satisfies c(X,Z) and (z,w) satisfies C(Z,Y)
    - Stronger than AC weaker than PC

k-consistency

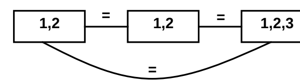
- Is there a common formalism for AC and PC?
  - AC: a value is extended to another variable
  - PC: a pair of values is extended to another variable
  - … we can continue
- *Definition:* CSP is k-consistent iff any (locally) consistent assignment of (k-1) different variables can be extended to a consistent assignment of one additional variable.



4-consistent graph

Strong k-consistency

- *Definition:* CSP is strongly k-consistent iff it is j-consistent for every j≤k.
- Visibly: strong k-consistency ⇒ k-consistency
- Moreover: strong k-consistency ⇒ j-consistency ∀j≤k
- In general: ¬ k-consistency ⇒ ¬ strong k-consistency
- NC = strong 1-consistency = 1-consistency
- AC = (strong) 2-consistency
- PC = (strong) 3-consistency

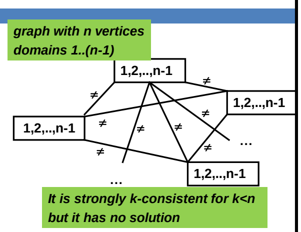


3-consistent graph

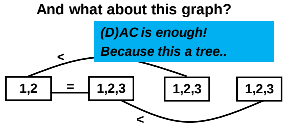
but not 2-consistent graph!

What *k-consistency* is enough?

- Assume that the number of vertices is *n*. What level of consistency do we need in order to find a solution?
- Strong n-consistency for graphs with *n* vertices!
  - n-consistency is not enough
  - strong k-consistency where k<n is not enough as well



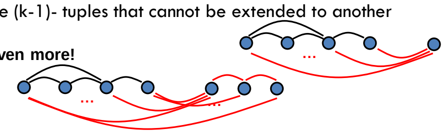
And what about this graph?



(i,j)-consistency

- k-consistency extends instantiation of (k-1) variables to a new variable,
- we remove (k-1)- tuples that cannot be extended to another variable.

We can do even more!



***Definition:* CSP is (i,j)-consistent iff every consistent instantiation of *i* variables can be extended to a consistent instantiation of any *j* additional variables.**

CSP is strongly (i,j)-consistent, iff it is (k,j)-consistent for every k≤i.

k-consistency	**= (k-1,1) consistency**
AC	**= (1,1) consistency**
PC	**= (2,1) consistency**

Inverse consistencies

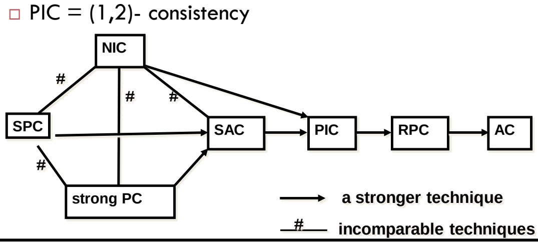
- Worst case time and space complexity of (i,j)-consistency is exponential in *i*, moreover we need to record forbidden *i*-tuples extensionally (see PC).
- What about keeping *i*=1 and increasing *j*?
- We already have such an example:
  RPC is (1,1)-consistency and sometimes (1,2)-consistency
- *Definition*: (1,k-1)-consistency is called k-inverse consistency.
- We remove values from the domain that cannot be consistently extended to additional (k-1) variables.
- Inverse path consistency (PIC) = (1,2)-consistency
- Neighbourhood inverse consistency (NIC) (Freuder , Elfe 1996)
- We remove values of *v* that cannot be consistently extended to the set of variables directly linked to *v*.

Singleton consistencies

- Key Idea: assign a value and make the rest of the problem consistent according to some consistency notion.
- *Definition*: CSP P is singleton A-consistent for some notion of A-consistency iff for every value *h* of any variable *X* the problem $P_{|X=h|}$ is A-consistent.
- *Features:*
  - \+ we remove only values from variable's domain - like NIC and RPC
  - \+ easy implementation
  - \- not so good time complexity
  - 1) singleton A-consistency ≥ A-consistency
  - 2) A-consistency ≥ B-consistency ⇒
    singleton A-consistency ≥ singleton B-consistency
  - 3) singleton (i,j)-consistency > (i,j+1)-consistency (SAC>PIC)
  - 4) strong (i+1,j)-consistency > singleton (i,j)-consistency (PC>SAC)

Consistency techniques at glance

- NC = 1- consistency
- AC = 2- consistency = (1,1)- consistency
- PC = 3- consistency = (2,1)- consistency
- PIC = (1,2)- consistency



Search+inference

How to solve the constraint problems?

- In addition to local search we have two other methods:
  - depth-first search
    - complete (finds a solution or proves its non-existence)
    - too slow (exponential)
      - explores "visibly" wrong valuations
  - consistency techniques
    - usually incomplete (inconsistent values stay in domains)
    - pretty fast (polynomial)
- Share advantages of both approaches - combine them!
  - label the variables step by step (backtracking)
  - maintain consistency after assigning a value

Solving techniques (1)

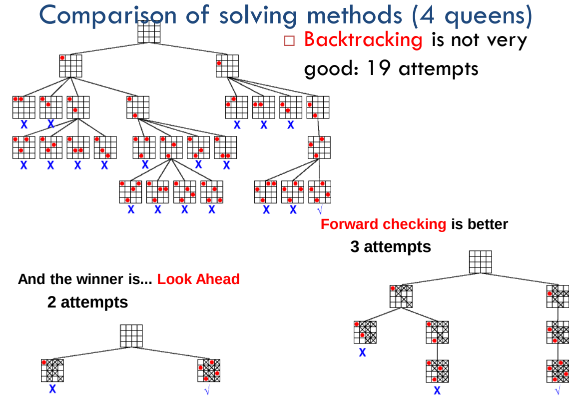
- Core procedure DFS:
  - assign variables one by one
  - ensure consistency after each assignment
- Look back:
  - maintain consistency among already assigned variables
  - look back= look to already assigned variables
  - if the consistency test return a conflict (+ explanation)
    - backtrack (basic) or
    - backjump

Solving techniques (2)

- Forward checking:
  - prevention technique
  - remove values from future variables which are incompatible with current assignments
  - check only future variables connected to some assigned variables by some constraint
- Partial look ahead
  - propagate the value assigned to the current variable to all future variables
  - DAC maintained in reverse order w.r.t. the labeling order (aka known as DAC look ahead)
  - it is not necessary to consider constraints involving past variables other than the current one
- Look Ahead
  - Like Partial Look Ahead but with AC instead of DAC
- MAC
  - AC performed initially
  - maintained after each assignment
- MCk:
  - Maintain strong-k-consistency
  - chronological backtracking

Comparison of solving methods (4 queens)

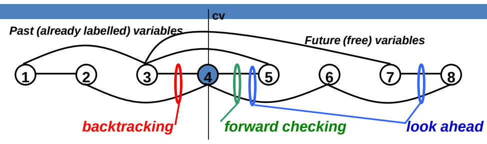
- Backtracking is not very good: 19 attempts



Forward checking is better
3 attempts

And the winner is... Look Ahead
2 attempts

Constraint propagation at glance



- Propagating through more constraints remove more inconsistencies (BT < FC < PLA < LA), of course it increases complexity of the step.
- Forward Checking does no increase complexity of backtracking, the constraint is just checked earlier in FC (BT tests it later).
- When using AC-4 in LA, the initialisation is done just once.
- Consistency can be ensured before starting search

Variable ordering(1)

- Variable ordering in labelling influence significantly efficiency of solvers (e.g. in tree-structured CSP).
- FIRST-FAIL principle
  - *„select the variable whose instantiation will lead to failure“*
  - it is better to tackle failures earlier, they can be become even harder
  - prefer the variables with smaller domain (dynamic order)
    - a smaller number of choices ~ lower probability of success
    - the dynamic order is appropriate only when new information appears during solving (e.g., in look ahead algorithms)

Variable ordering(2)

- *„solve the hard cases first, they may become even harder later“*
  - prefer the most constrained variables
    - it is more complicated to label such variables (it is possible to assume complexity of satisfaction of the constraints)
    - this heuristic is used when there are domains of equal size
  - prefer the variables with more constraints to past variables
    - a static heuristic that is useful for look-back techniques

Value ordering (1)

- Order of values in labelling influence significantly efficiency (if we choose the right value each time, no backtrack is necessary).
- What value ordering for the variable should be chosen in general?
- SUCCEED FIRST principle
  - „prefer the values which have a better chance of belonging to the solution“
  - if they all look the same then we have to check all values

Value ordering (2)

- SUCCEED FIRST does not go against FIRST-FAIL !
  - prefer the values with more supporters
    - this information can be found in AC-4
  - prefer the value leading to less domain reduction
    - this information can be computed using singleton consistency
  - prefer the value simplifying the problem
    - solve approximation of the problem (e.g. a tree)
- Generic heuristics are usually too complex for computation.
- It is better to use problem-driven heuristics that proposes the value!

Other interesting issues

- Soft constraints
  - the world is not black and white
  - satisfaction relaxed to degrees of satisfaction
  - a tuple satisfies a constraint to certain degree
  - this degree may represent a preference or a cost
  - satisfaction problem → optimization problem
  - find not just a solution but the best solution
- Global constraints
  - Specific constraints that occur often in practice, and specific efficient propagation algorithms for them
- Symmetry breaking

LP formulation of CSPs

CSP	LP
Constraint	Set of facts defining a predicate
CSP: set of constraints	LP program : set of predicate definitions
Satisfying the CSP	Clause with - body: all the predicates - head: contains all the variables of the CSP
Finding a solution	Executing a goal matching the head of the clause

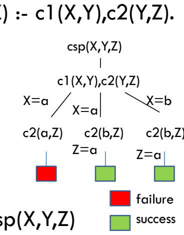
Example: CSP → LP program

CSP

- Variables: x,y,z
- Domain {a,b,c} for all the variables
- Constraints:
  - $c_1(x,y)=\{(a,a),(a,b),(b,b)\}$
  - $c_2(y,z)=\{(b,a)\}$
- Solutions:
  - (X=a, Y=b, Z=a),
  - (X=b, Y=b, Z=a),

LP

csp(X,Y,Z) :- c1(X,Y),c2(Y,Z).
c1(a,a).
c1(a,b).
c1(b,b).
c2(b,a).



Goal: csp(X,Y,Z)

LP formulation of CPS (2)

- Summarizing:
  - a finite domain CSP= LP program with one clause and several facts
- LP can represent much more complex things
  - recursion
  - function symbols
- Functions can be used for a more compact representation of constraints

Examples: CSP → LP program

- Expressing binary constraint eq(X,Y): X=Y
- Enumerating all facts…not the way to go
- just one fact: eq(X,X).

- Expressing binary constraint neq(X,Y): X≠Y
- just one clause and one fact:
  - neq(X,X):- !, fail.
  - neq(X,Y).
- fail built in predicate that always fails
- ! cut: makes sure second clause in not tried if first fails

LP formulation of CSPs(4)

- LP solution engine corresponds to depth-first search with chronological backtracking
  - not the most efficient way to solve CSPs

  - → Constraint Logic Programming
  - extends LP allowing for the use of CP techniques for improving solving
  - extend CP by allowing more general and compact definition of constraints (formulas over a specific language)

Constraint Logic Programming

CLP = CP + LP

- CLP : the merger of two declarative paradigms
  - Constraint solving
  - Logic Programming
- Common base: mathematical relations

Key feature

- Combing logic and solving in an algorithmic context
- Conceptual model of a problem: its precise formulation in logic
- Design model of a problem: its algorithmic formulation, sequence of steps for solving it
- CLP can express both models
- Provides mapping: conceptual models → design models

Example (..seen in CP)

- Cryptoarithmetic problem:
- SEND+MORE=MONEY
- Conceptual model:

New predicate/constraint definition:
name: smm
arguments: S,E,N,D,M,O,R,Y
the variables of the problem

```
smm(S,E,N,D,M,O,R,Y) :-
    [S,E,N,D,M,O,R,Y] :: 0 . . 9,
              1000 * S + 100 * E + 10 * N + D
            +1000 * M + 100* O + 10 * R + E
    #= 10000 * M +1000 * O + 100 * N + 10 * E + Y,
    M #> 1, S#>1,
    alldifferent([S,E,N,D,M,O,R,Y])
```

Body of rule: defines the new predicate/ constraint in terms of other (known) predicates/ constraints

ECLiPSe notation

Example(2)

- LABELING (basic CLP search procedure)

```
labeling([ ]).
labeling([V|Rest]) :-
                    indomain(V),
                    labeling(Rest).
```

Built-in predicate allows to nondeterministically set the value of V to each of its possible values in turn

- Design model of SEND+MORE=MONEY
  - smm(S,E,N,D,M,O,R,Y), labeling([S,E,N,D,M,O,R,Y])
  - underlying finite domain constraint solver
- Returned solution:
- S=9, E= 5, N= 6, D= 7, M= 1, O= 0, R= 8, Y= 2.

Important features of CLP

- The CLP paradigm is generic in
  - the choice of primitive constraints
  - the choice of the underlying constraint solver
- → **CLP Scheme**

- In our cryptoarithmetic example
  - Primitive constraints (needed):
    - bounded integer constraints
  - Possible underlying solvers:
    - propagation based
    - mixed integer programming (MIP)
    - local search-based

A little bit of history

- CLP was developed by three independent research teams:
  - Colmerauer et al. in Marseilles (France)
  - Jaffar and Lassez et al. in Melbourne (Australia)
  - Dincbas et al. Munich (Germany)

  - CLP as a generalization of LP
    - Primitive constraints: only syntactic equality
    - Solver: unification

A little bit of history (2)

- Research development lines:

  - generalizing unification to other types of equality

  - allowing more flexible dynamic evaluation
    - relaxing Prolog's left-to-right literal selection strategy allowing goals to be delayed until sufficiently instantiated

A little bit of history (3)

- The Melbourne group
  - coined CLP term 1986
  - schema and semantics for CLP languages
  - CLP(R) = Prolog + arithmetic constraints
    - Solver: incremental Simplex
  - Applications: financial and engineering
- The Marseilles group
  - Prolog II (early 80's):
    - first logic programming language with constraints
    - equations and disequations over rational trees
  - Prolog III (late 80's)
    - constraints over Booleans
    - linear arithmetic over rational numbers
    - constraints over lists
  - Applications: chemical reasoning

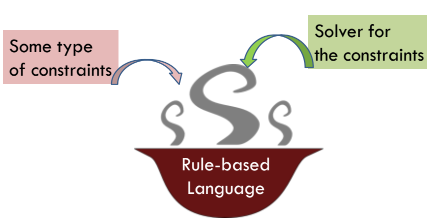
A little bit of history (4)

- The Munich group:
  - CHIP language
    - Prolog's backtracking search + AI consistency techniques
    - Finite domain constraints
  - Applications: circuit diagnosis

A little bit of history (5)

- From black box to glass box
  - languages that allow programmers to extend and/or define new underlying solvers
- Hybrid constraint-solving techniques combining
  - propagator-based solving + linear programming
  - MIP + local search
  - ECLiPSe

CLP Scheme (Jaffar and Lassez '87)



Different ingredients, different soup!

CLP scheme: key idea

- Key idea
  - Parameterize:
    - operational semantics
    - declarative semantics
    - relation between the two
  - by a choice of
    - constraints
    - solver for the constraints
    - algebraic and logical semantics for the constraints

The Constraint Domain(1)

- CLP schema defines the class of languages CLP(C), parametric in C
- **C : constraint domain**, definition and interpretation of built-in primitive constraints and functions
  - Constraint domain **signature S_C**
    - set of function and predicate symbols
    - map symbol → arity
    - Thus defines the terms of the language
      - variables
      - function terms $f(t_1,\ldots,t_n)$ f function symbol and t_i term
  - Class of **constraints L_C**
    - predefined subset of first order S_C-formulas
  - **Domain of computation D_C**
    - set D
    - mapping:
    - function symbols in signature S_C → functions over D
    - predicate symbols in signature S_C → relations over D
      - respecting the arities
    - algebraic semantics of the constraints

The Constraint Domain(2)

- **Constraint Theory T_C**
  - (possibly infinite) set of closed S_C-formulae
  - logical semantics of the constraints
- **Solver $solv_C$**
  - mapping
  - constraints → {true,false,unknown}
  - $solve_C(c)$=true means "c is satisfiable"
  - $solve_C(c)$=false means "c is not satisfiable"
  - $solce_C(c)$= unknown means "don't know if it satisfiable or not"
  - operational semantics of constraints
- Note
  - **Primitive constraint**: atom $p(t_1,\ldots,t_n)$ in L_C
  - **Constraint**: first order formula built from primitive constraints in L_C

Assumptions

- **Equality**
  - binary predicate symbol "=" is in S_C
  - = interpreted as the identity relation in D_C
  - standard equality axioms in T_C
- **L_c always contains**
  - all atoms with predicate symbol =
  - *true* (the "always true" constraint)
  - *false* (the "always false" constraint)
- **D_C, $solv_C$ and T_C agree**
  - D_C is a model of T_C
  - for any primitive constraint c
    - if $solv_C(c)$=false then $T_C \models \neg\tilde{\exists}c$
    - if $solv_C(c)$ =true then $T_C \models \tilde{\exists}c$

Example: the constraint domain *Real*

- Signature S_C:
  - predicate symbols: <,>,=,≤,≥
    - all binary
  - function symbols:
    - Binary: +,*,-,/
    - Constants: sequences of digits possibly with a decimal point (1, 2.3…)
- Constraints L_C
  - primitive constraints: <,>,=,≤,≥
- Domain of computation D_C
  - domain: set of real numbers R
  - <,>,=,≤,≥ → usual arithmetic relations
  - +,*,-,/ → usual arithmetic functions over R
  - 1,2,4.5…→decimal representation of elements of R
- Theory T_C
  - Theory of real closed fields
- Solver $solv_C$
  - Simplex + Gauss-Jordan elimination
- Corresponding CLP language : CLP(R)

Example: the constraint domain *Term*

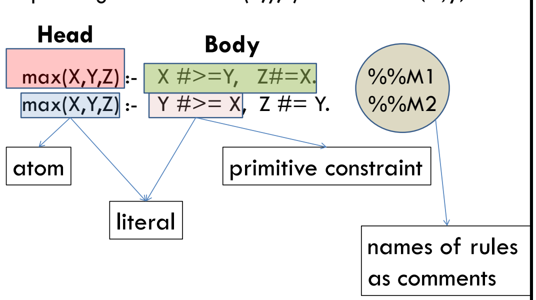
- Signature S_C:
  - predicate symbols: = binary
  - function (and constant) symbols:
    - strings of alphanumeric characters
- Constraints L_C
  - primitive constraints: =
- Domain of computation D_C
  - domain: set of finite trees *Tree*
  - = → identity relation over Tree
  - Interpretation of n-ary function f:
  - $I(f) : T^n \rightarrow T$, n trees ↦ tree with root f and the n-trees as children
- Theory T_C
    - Clark's theory for Term (= syntactic equality)
- Solver $solv_C$
    - unification algorithm
- Corresponding CLP language : CLP(Term), aka Prolog

Syntax of Constraint Logic Programs

- Constraint logic programs are sets of logical statements (aka rule or clauses) which extend a constraint domain by defining new constraints in terms of primitive constraints
- **Constraint logic program** = set of rules
- **Rule** H :- B
  - H, **head** of the rule, is an atom
  - B, **body** of the rule, finite sequence of literals
  - ☒ the empty sequence
  - H:-☒, written H. for short
- **Literal:** atom or primitive constraint
- **Atom:** $p(t_1,\ldots,t_n)$ p predicate symbol, t_i term

Example

- Expressing Relation: max(x,y,z) ⟷ z=max(x,y)



Operational Semantics (1)

- Operational semantics provides a way of repeatedly unfolding a goal with user-defined constraints until a conjunction of primitive constraints is reached
- **Renaming:** bijective mapping between variables
- **Syntactic object:** formula, rule or constraint
- **Variants** : synt. objs. s and s' are variants iff there exists a renaming r such that r(s)=s'
- Definition of User-defined predicate p in program P **$defn_P(p)$:**
  - set of of rules in P with head of the form $p(s_1,\ldots,s_n)$
  - renaming assumption: every time $defn_P(p)$ is called it returns a variants with distinct new variables

Operational semantics (2)

- **State <G|c>**
  - **G** current **goal** (current literal sequence $L_1,\ldots,L_m$)
  - **c** current **constraint store** (conjunction of primitive constraints)
- **Reduction step** from state S to state S' (S→S')
  - if left-most literal L_1 is a primitive constraint
    - if solv(c $\wedge$ L_1)$\neq$false
    - then S'=<L_2, …, L_m | $L_1 \wedge$ c>
    - else S' =< ⊠| false>
  - if left-most literal L_1 is an atom of form $p(s_1,\ldots,s_n)$
    - if $defn_P(p) \neq \emptyset$
    - then S' =<$s_1=t_1,\ldots, s_n = t_n$,B,$L_2,\ldots,L_m$|c> , for some (A :- B) $\in defn_P(p)$ with A of form $p(t_1,\ldots,t_n)$
    - else S' = <⊠ |false>

Operational semantics(3)

- **Derivation from a goal G in a program P:** sequence of states S_0 → S_1→ … → S_n, where
  - S_0=<G|true>
  - S_{i-1} →S_i reduction using rules is P
- **Length** of derivation S_0→S_1→…→S_n : n
- A derivation is **finished** when the last goal cannot be reduced
- Last state of a finished derivation: < ⊠|c>
  - if c = false, **failing derivation**
  - otherwise **successful derivation**
- **Answers of a goal G for a program P**
  - constraints $\exists_{vars(G)}c$ where there is a successful derivation from G with final state with constraint store c

Example(1)

<max(A,B,C), B #=2 | true>

max(X,Y,Z) :- X #>= Y, Z #= X

<A = X, B = Y, C = Z, X #>= Y, Z #=X, B#=2 |true>

<B = Y, C = Z, X #>= Y, Z #=X, B#=2 | A = X >

<C = Z, X #>= Y, Z #=X, B#=2 | A = X ∧ B = Y >

<X #>= Y, Z #=X, B#=2 | A = X ∧ B = Y ∧ C = Z>

< Z #=X, B#=2 | A = X ∧ B = Y ∧ C = Z ∧ X ≥Y>

< B#=2 | A = X ∧ B = Y ∧ C = Z ∧ X ≥Y ∧Z =X>

< ⊠| A = X ∧ B = Y ∧ C = Z ∧ X ≥Y ∧Z =X ∧B=2 >

projecting onto the variables of the original goal gives $A\geq 2 \wedge B=2 \wedge C=A$

Example(2)

First Derivation

<A#=1,max(1,2,1)| true>

<max(A,2,1)|A=1>

max(X,Y,Z) :- X #>= Y, Z #= X

<A= X, 2 = Y, 1=Z, X #>= Y, Z #=X| A = 1 >

< 2 = Y, 1=Z, X #>= Y, Z #=X| A = 1 ∧ A=X >

<1=Z, X #>= Y, Z #=X| A = 1 ∧ A=X ∧2=Y >

< X #>= Y, Z #=X| A = 1 ∧ A=X ∧2=Y ∧1=Z >

< ⊠| false>

Second Derivation

<A#=1,max(1,2,1)| true>

<max(A,2,1)|A=1>

max(X,Y,Z) :- Y #>= X, Z #= Y

<A= X, 2 = Y, 1=Z, Y#>= X, Z #=Y| A = 1 >

< 2 = Y, 1=Z, Y #>= X, Z #=Y| A = 1 ∧ A=X >

<1=Z, Y #>= X, Z #=Y| A = 1 ∧ A=X ∧2=Y >

< Y #>= X, Z #=Y| A = 1 ∧ A=X ∧2=Y ∧1=Z >

< ⊠| false>

Fails!

Example(3)

CLP(R) program
factr(0,1). %%R1
factr(N,N*F):- N #>=1, factr(N-1,F). %%R2

Failed derivation

<factr(2,1) | true>
⇓ R1
<2= 0, 1= X | true>
⇓
< ⊠ | false>

Successful derivation

<factr(1,X) | true>
⇓ R2
<1 = N, X = N*F, N #>= 1, factr(N-1,F) | true>
⇓
< X = N*F, N #>= 1, factr(N-1,F) | 1 = N >
⇓
< N #>= 1, factr(N-1,F) | 1 = N ∧ X = N*F >
⇓
< factr(N-1,F) | 1 = N ∧ X = N*F∧ N ≥ 1>
⇓ R1
< N-1=0, F=1 | 1 = N ∧ X = N*F∧ N ≥ 1 >
⇓
< F=1 | 1 = N ∧ X = N*F∧ N ≥ 1∧ N-1=0 >
⇓
< ⊠ | 1 = N ∧ X = N*F∧ N ≥ 1∧ N-1=0∧F=1 >

projecting onto the variables of the original goal gives X=1

Role of the solver(1)

- Check if L∧c is satisfiable, knowing that c was satisfiable
  - → incremental constraint solving!
- The solver may be incomplete
  - something may be unsatisfiable and the solver may not detect this
  - gives pseudo-answers
  - Identify a class of goals for which the solver is known to be complete

<Y= X * X, Y#<0 | true>
⇓
<Y#<0 | Y=X*X>
⇓
< ⊠ | Y=X*X ∧ Y<0>

successful derivation for the CLP(R) solver

Operational semantics confluence(1)

- Sources of non-determinism in derivations
  1. choice of rule
  2. choice of renaming
  3. choice of literal

1. Different rules → (possibly) different answers
   - For completeness, all rule must be considered
2. Renaming is harmless
   - the solver does not take into account names of variables

Operational semantics confluence(2)

3. Independence from the choice of literal selection

- **Literal selection strategy:** given a derivation, returns a literal in the last goal
  - may select different literals in same goal if occurring more than once in the derivation
- **Derivation** is **via a literal selections strategy** S iff all choices are performed through S

When literal selection may cause trouble

- Literal selection influences the order of the constraints in the constraint store
- such order may be crucial for the solver
- Example 1:
  - CLP(R) program : p(Y):- Y#=1, Y#=2.
  - Solver: ignoring the last primitive constraint in its argument
    - solv(X=Y) → unknown
    - solv(X=Y ∧Y=1) → unknown
    - solv(X=Y ∧Y=1 ∧ Y=2) → unknown
    - solve(Y=2) → unknown
    - solve(Y=2 ∧Y=1) → unknown
    - solve(Y=2 ∧Y=1 ∧ X=Y) → false
  - left-to-right for goal p(X) : ∃Y(X=Y ∧Y=1 ∧Y=2) (unknown)
  - right-to-left for goal p(X): ∃Y(Y=2 ∧Y=1 ∧X=Y) false

When literal selection may cause trouble

- Example 2:
  - CLP(R) program : p(Y):- Y#=1, Y#=2.
  - Solver: complete for all constraints with only 2 primitives, unknown to all others
    - solv(X=Y) → true
    - solv(X=Y ∧Y=1) → true
    - solv(X=Y ∧Y=1 ∧ Y=2) → unknown
    - solve(Y=2) → true
    - solve(Y=2 ∧Y=1) → false
    - solve(Y=2 ∧Y=1 ∧ X=Y) → unknown
  - left-to-right for goal p(X): ∃Y(X=Y ∧Y=1 ∧Y=2) unknown
  - right-to-left for goal p(X):∃Y(Y=2 ∧Y=1) fails
  - Not monotonic

Well-behaved solvers

- Solver solv is **well-behaved** for constraint domain C if for any constraints c and c' in L_C it is:
- **Logical:** solv(c) = solve(c') whenever ⊨c ↔c'
  - if the two constraints are logically equivalent independently of the constraint domain, then the solver answers the same for both
- **Monotonic:** if solv(c)= false then solve(c')=false whenever ⊨c $\leftarrow\exists_{vars(c)}$ c'
  - if the solver fails c then whenever c' contains more constraints it fails also c'
- Misbehavior Example 1: not logical
- Misbehavior Example 2: not monotonic
- Any complete solver is well-behaved

Independence of literal selection strategy

- **Switching Lemma:**
- Let
  - S state
  - L and L' literals in the goal of S
  - solv well-behaved solver
  - S→S1→ S' non-failed derivation obtained by solv with L selected first followed by L'
- Then
  - there is a derivation S → S2→ S'' obtained by solv with L' selected first followed by L
  - S' and S'' are identical up to reordering of their constraint components
- **TH:** Let
  - solv well behaved solver
  - P program
  - G goal
  - there is a derivation from G with answer c
- Then
  - for any literal selection strategy S
  - there is a derivation of the same length form G via S with answer a reordering of c

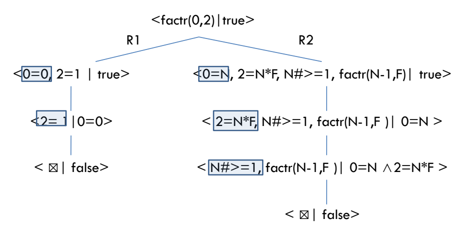
Derivation tree

- Independence of literal selection → the solver can use a single selection strategy
- Single strategy → collect all derivations in a single tree
- **Derivation tree** for goal G, program P and and selection strategy S
  - node: states
  - root: <G|true>
  - children of a node with state s1: states reachable from s1 given strategy s
  - different branches : different rules
- unique up to variable renaming
- derivation: path from root to leaf
  - successful: <⊠|c> c not false leaf
  - failed: <⊠|false> leaf

Example of derivation tree

```
CLP(R) program
factr(0,1).                                   %%R1
factr(N,N*F):-   N #>=1, factr(N-1,F).   %%R2
```

Left-most strategy



Possible outcomes of an execution

- The execution of a CL program can return:
  - yes and an answer (obtained from the constraint store of the leaf in the derivation tree)
  - no
- A goal G **finitely fails** if
  - it has a finite set of derivations
  - they all fail
- Example: factr(0,2) finitely fails
- Finite failure is NOT independent of the literal choice, even if the solver is well-behaved
- **Fair selection strategy S:** in every infinite derivation via S each literal in the derivation is selected
- Example
  - left-to right: unfair
  - oldest first: fair
- **TH:** If the solver is well-behaved then finite failure is independent of fair selection strategies

The semantics of success

- Each rule corresponds to a formula:

$$A :- \quad L_1,\ldots,L_n \Rightarrow \tilde{\forall}(A \leftarrow L_1 \wedge \ldots \wedge L_n)$$

rule of a CLP program → corresponding implication

```
max(X,Y,Z) :-   X #>=Z,   Z#=X.
max(X,Y,Z) :-   Y #>= X,  Z #= Y.
```

$$\forall X \forall Y \forall Z.\max(X,Y,Z) \leftarrow (X \geq Y \wedge Z = X)) \wedge$$
$$\forall X \forall Y \forall Z.\max(X,Y,Z) \leftarrow (Y \geq X \wedge Z = Y))$$

CLP program → conjunction of implications

- Logical semantics of a CLP(C) program P
  - theory obtained adding (the formulas corresponding to) the rules of P to the constraint theory T_C of the constraint domain

Logical Soundness and completeness(1)

- It is desirable for the operational semantics to be sound w.r.t. the logical semantics
- Soundness: the answers returned by the operational semantics logically imply the initial goal
- Thus, "goal G has answer c" means "if c holds, so does G "

Logical Soundness of the semantics of success

- **Logical soundness**
- Let:
  - T_C: constraint theory for constraint domain C
  - P: CLP(C) program
  - G goal with answer c
- then $P, T_C \models c \rightarrow G$

Algebraic semantics for success

- Find a model for the program that is the intended interpretation of the program
- Agree with the interpretation of the primitive constraint and function symbols in constraint domain
- Extend the interpretation to all user-defined predicate symbols in P
- A **C-interpretation** of a CLP(C) program P, is an interpretation that agrees with the domain of computation D_C on the symbols in S_C
- **C-base$_P$** $=\{p(d_1,\ldots,d_n) \mid p$ n-ary user-defined predicate in P and d_i domain element of $D_C\}$
- C-interpretation identified by the subset of the C-base$_P$ which it makes true
- A **C-model** of a CLP(C) program P is a C-interpretation which is a model of P
- lm(P,c): least (under subset ordering) C-model of a program P
  - always exists
  - usually chosen as "inteded" representation since it is the most conservative
  - same as least Herbrand model as algebraic semantics for logic programs

Example of least model

CLP(R) program

```
factr(0,1).                          %%R1
factr(N,N*F):-   N #>=1, factr(N-1,F).  %%R2
```

Has an infinite number of real models, e.g.,
model 1: $\{factr(n,n!) \mid n \in \{0,1,2,\ldots\}\} \cup \{factr(n,0) \mid n \in \{0,1,2,\ldots\}\}$
model 2: $\{factr(n,n!) \mid n \in \{0,1,2,\ldots\}\}$
model 3: $\{factr(r,r') \mid r \in R\}$

The least model is model 2

Role of the least model

- If a goal is satisfiable in the least C-model it is so in all models
- TH: let
  - P CLP(C) program
  - G goal
  - σ valuation
- then
  - $P,D_C \models_\sigma G$ iff $lm(P,C) \models_\sigma G$

- TH: let
  - P CLP(C) program
  - G goal
- then
  - $P,D_C \models \tilde{\exists}G$ iff $lm(P,C) \models \tilde{\exists}G$

Algebraic Soundness of the semantics of success(1)

- Soundness w.r.t. the algebraic semantics: the operational semantics only answers which are solutions of the goal
- Let
  - P , CLP(C) program
  - G goal with answer c
- then $lm(P,C) \models c \rightarrow G$

Completeness of success semantics

- Algebraic and logical soundness ensure that the operational semantics only returns answers which are solutions of the goal
- Completeness: the operational semantics returns all the solutions of a goal

Logical Completeness of success

- **Logical completeness:** the answers returned by the operational semantics cover all of the constraints which imply the goal
- Let:
  - T_C: constraint theory for constraint domain C
  - P: CLP(C) program
  - G goal, c constraint such that $P,T_C \models c \rightarrow G$
- then G has answers $c_1,\ldots,c_n$ such that

  $T_C \models c \rightarrow (c_1 \vee \ldots \vee c_n)$

Logical completeness of success

- Notice that more than one answer may be needed to cover c (i.e. n>1 in some cases)

- **Example:**
- CLP(R) program:
  - p(X):- X#>=2.
  - p(X):- X#<=2.
- Consider Goal p(X)
- Then $P,T_{Real} \models true \rightarrow p(X)$
- p(X) has answers $c_1=(X\geq 2)$ and $c_2=(X\leq 2)$
- Both are needed to cover true
- $T_{Real} \models true \rightarrow (c_1 \vee c_2)$

Algebraic completeness

- In order to show that the operational semantics is complete w.r.t. the algebraic semantics we need to introduce an additional semantics for CLP programs that bridges the gap between the algebraic and the operational semantics

Fixed Point Semantics(1)

- Based on the immediate consequence operator
  - set of facts in a C-interpretation → set of facts implied by the rules in the program
  - captures Modus Ponens
- Generalizes the T_P semantics for logic programs

- **Immediate consequence function** T_P^C for CLP(C) program P:
  - I: C-interpretation of P
  - σ: range over valuations for C
  - then $T_P^C(I)=\{\sigma(A) \mid A{:}\text{-}L_1,\ldots,L_n \text{ rule in P s.t. } I\models_\sigma L_1\wedge\ldots\wedge L_n\}$

Fixed Point Semantics(2)

- Notice that:
- $I\models_\sigma L_1\wedge\ldots\wedge L_n$ iff
  - for each literal L_i
  - either L_i primitive constraint s.t. $C \models_\sigma L_i$
  - or L_i user-defined predicate $p(t_1,\ldots,t_m)$ such that $p(\sigma(t_1),\ldots,\sigma(t_m))\in I$
  - T_P^C is continuous and monotonic on the complete lattice $\mathcal{P}(\text{C-base}_P)$
  - → it has a greatest and a least fixed point, $gfp(T_P^C)$ and $lfp(T_P^C)$.

Fixed Point semantics

- Kleene's fixpoint theorem
  - the least fixpoint of F is the supremum of the ascending Kleene chain of F
    - $\bot \leq F(\bot) \leq F(F(\bot)) \leq \ldots \leq F^n(\bot) \leq \ldots$
    - $lfp(F) = \sup \{F^n(\bot) \mid n \in N\}$
  - the greatest fixed point of F is the infimum of the descending Kleene chain
    - $\top \geq F(\top) \geq F(F(\top)) \geq \ldots \geq F^n(\top) \geq \ldots$
    - $gfp(F) = \inf (\{F^n(\bot) \mid n \in N\}$

C-models of a program P and T_P^C

- Lemma: M C-model of program P iff M is a pre-fixpoint of T_P^C, that is, $T_P^C(M) \subseteq M$
- **Main result:**
- let
  - P, CLP(C) program
- then $Im(P,C) = lfp(T_P^C)$

Algebraic completeness of the semantics of success

- **Algebraic completeness:** the answers provided by the operational semantics cover all solutions to the goal
- **TH:** Let
  - P , CLP(C) program
  - G goal
  - θ evaluation such that $Im(P,C) \models_\theta G$
- then G has answer c such that $D_C \models_\theta c$
- The proof uses $Im(P,C) = lfp(T_P^C)$
- **Soundess+Completeness:** The solutions of the goal in the minimal model are exactly the solutions to the constraints the operational semantics returns as answers
- **TH:** Let
  - P , CLP(C) program
  - G goal with answers $c_1, c_2, \ldots$
- Then $Im(P,C) \models G \leftrightarrow \bigvee_{i=1,\ldots,\infty} c_i$

Semantics for finite failure

- A goal G can finitely fail
- the semantics for success does not work well with finite failure
- In fact, there is always a C-model, the entire C-base, in which every constraint is satisfiable
- → new semantics based on the Clark completion
  - captures the if-and-only-if nature of rules for defining predicates
  - rules should cover all the cases which make the predicate true

Clark completion

- The **definition** of n-ary predicate symbol p in the program P is the formula:
  - $\forall X_1 \ldots \forall X_n\ p(X_1,\ldots,X_n) \leftrightarrow B_1 \vee \ldots \vee B_m$
  - where each B_i
    - corresponds to a rule $p(t_1,\ldots,t_n)$:-$L_1,\ldots,L_k$
    - is of the form
    - $\exists Y_1 \ldots \exists Y_j\ (X_1=t_1 \wedge \ldots \wedge X_n=t_n \wedge L_1 \wedge \ldots \wedge L_k)$
    - $Y_1 \ldots Y_j$ variables in the original rule
    - $X_1,\ldots,X_n$ variables that do not appear in any rule
  - If there is no rule with head p, we have
  - $\forall X_1 \ldots \forall X_n\ p(X_1,\ldots,X_n) \leftrightarrow false\ (\vee \emptyset)$
- **Clark-completion P***of a CLP program P: conjunction of all the definitions of the user defined predicates in P

Example of Clark Completion(1)

- CLP program P:

```
max(X,Y,Z) :-   X #>=Y,   Z#=X.
max(X,Y,Z) :-   Y #>= X,  Z #= Y.
```

- Clarke-completion P* of P:

$\forall P \forall Q \forall R\ max(P,Q,R) \leftrightarrow \exists X \exists Y \exists Z(P=X \wedge Q=Y \wedge R=Z \wedge X \geq Y \wedge Z=X) \vee \exists X \exists Y \exists Z(P=X \wedge Q=Y \wedge R=Z \wedge Y \geq X \wedge Z= Y)$

- max(1,2,1) is a goal which finitely fails
- its negation is implied by the Clark completion

Example of Clark Completion(2)

- CLP program P:

```
factr(0,1).                                %%R1
factr(N,N*F):-   N #>=1, factr(N-1,F).  %%R2
```

- Clark-completion P* of P:

$\forall X \forall Y\ factr(X,Y) \leftrightarrow (X=0 \wedge Y=1) \vee$
$\exists N \exists F(X=N \wedge Y=N*F \wedge N \geq 1 \wedge factr(N-1,F))$

- factr(0,2) is a goal which finitely fails
- its negation is implied by the Clark completion

Models of a Clark completion

- Clark completion P* of program P captures the true meaning of a program
- Thus, intended interpretation of a P is a C-interpretation which is a model for P*.
- There may be more than one C-model for the Clark completion

Example of models of the Clark completion

program CLP(R) P

factr(0,1). %%R1

factr(N,N*F):- N #>=1, factr(N-1,F). %%R2

Clark completion P* of P

$\forall X \forall Y$ factr(X,Y) $\leftrightarrow$ (X=0 $\wedge$Y=1) $\vee$

$\exists N \exists F$(X=N $\wedge$ Y=N*F $\wedge$ N $\geq$ 1 $\wedge$ factr(N-1,F))

Has an infinite number of real-interpretations, e.g.,

I1: {factr(n,n!) | n$\in${0,1,2,…} }$\cup${factr(n,0)| n$\in${0,1,2,…} }

I2: {factr(n,n!) | n$\in${0,1,2,…}}

I3: {factr(r,r')| r $\in$R}

Only I2 is a R-model of the Clark completion

I1,I2 and I3 are all R-models given the semantics of success

Clark-completion and fixed points

- **TH:** Let
  - P CLP(C) program
  - P* Clark-completion
  - T_P^C immediate consequence operator
- then
  - **I is a model of P* iff it is a fixpoint of T_P^C**
- **Relation between the algebraic semantics of the completion and the fixpoint semantics**
- **TH:** Let
  - P, P*, T_P^C as above
  - gm(P*,C) the greatest C-model of P*
  - lm(P*,C) the least C-model of P*
- Then
  - **lm(P*,C)=lfp(T_P^C)=lm(P,C)**
  - **gm(P*,C)=gfp(T_P^C)**

Modeling success and failure

- The semantics based on the Clark-completion allows to model success
  - TH: Let
    - T_C: constraint theory of constraint domain C
    - P: CLP(C) program
    - G goal
  - Then
    - **P*,$T_C \models \tilde{\exists}$G iff lm(P*,C) $\models \tilde{\exists}$G iff lm(P,C) $\models \tilde{\exists}$G iff P,$T_C \models \tilde{\exists}$G**
- The semantics based on the Clark-completion allows to model failure
  - TH: Let
    - T_C: constraint theory of constraint domain C
    - P: CLP(C) program
    - G goal
  - Then
    - P*,$T_C \models \neg\tilde{\exists}$G iff gm(P*,C) $\models \neg\tilde{\exists}$G

Results for the semantics of success continue to hold

1. TH: Let P be a CLP(C) program. Then $T_C \models$P*$\rightarrow$ P
2. TH: Let P be a CLP(C) program. Then P,$T_C \models$c$\rightarrow$ G then P*,$T_C \models$c$\rightarrow$ G
3. TH: Let P be a CLP(C) program, G goal with answer c. P*,$T_C \models$c$\rightarrow$ G
4. TH: Let P be a CLP(C) program. Then P*,$T_C \models$c$\rightarrow$ G then P,$T_C \models$c$\rightarrow$ G
5. TH: Let P be a CLP(C) program, G a goal and c a constraint. If P*,$T_C \models$c$\rightarrow$ G then G has answers $c_1,\ldots, c_n$ such that $T_C \models$c$\rightarrow$ ($c_1 \vee\ldots\vee c_n$)

Logical soundness for finite failure

- **Finitely evaluable goal**: it has no infinite derivations
- TH.: Let
  - T_C theory
  - P CLP(C) program
  - G finitely evaluable goal with answers $c_1,\dots,c_n$.
- Then
  - $P^*,T_C \models G \leftrightarrow (c_1 \vee \dots \vee c_n)$
- TH. (special case of the one above when there are no answers) Let
  - T_C theory
  - P CLP(C) program
  - G finitely failing goal
- Then
  - $P^*,T_C \models \neg\tilde{\exists}G$

Algebraic soundness of finite failure

- Follows immediately from logical soundness for finite failure since any intended interpretation of the constraint domain is a model of the constraint theory
- TH: Algebraic soundness
- Let
  - P CLP(C) program
  - G finitely failing goal
- Then
  - $P^*,D_C \models \neg\tilde{\exists}G$ and
  - $gm(P^*,C) \models \neg\tilde{\exists}G$

Logical completeness of finite failure

- Additional assumptions
  - **theory-complete** solver
  - **fair** literal selections strategy
- THM (logical completeness) Let
  - T_C: constraint theory of constraint domain C
  - P: CLP(C) program
  - G goal
- Then,
  - if $P^*,T_C \models \neg\tilde{\exists}G$
  - then G finitely fails

Algebraic completeness for finite failure: assumptions

- Solver should agree with the domain of computation on the satisfiability of constraints→ should be complete
- Complete solver→theory satisfaction-complete
  - satisfaction complete: able to determine for each constraint if it is satisfiable or not
- Completeness of the solver and fair literal selections not sufficient for algebraic completeness
- **Finitely evaluable** goal G for a program P: a goal with no infinite derivations

Example

- CLP(Term) program P
  - q(a):- p(X).
  - p(f(X)) :-p(X)
- Clark completion P*
  - $\forall Y(p(Y)) \leftrightarrow \exists X\ (Y=f(X) \wedge p(X)))\ \wedge \forall Y(q(Y) \leftrightarrow \exists X(Y=a\ \wedge p(X)))$
  - The only Term-model of P* is $\emptyset$ but q(a) does finitely fail with a complete solver for any selection rule

Algebraic completeness of finite failure

- **Finitely evaluable** goal G for a program P: a goal with no infinite derivations
- **THM(Algebraic completeness of Finite Failure)Let**
  - P, CLP(C) program
  - G finitely evaluable goal
  - solv complete solver
  - T_C satisfaction complete
  - fair selection strategy
- Then
  - If $Im(P^*,C) \models \neg\tilde{\exists}G$
  - then G finitely fails

Extended semantics

- Many extensions have been proposed
  - Negation
  - Optimization
  - …many others

CLP formulation of CSPs

- Formulation of standard CSPs (where constraints are represented by sets of allowed tuples) inherited from LP
- CLP provides
  - equality, disequality
  - standard mathematical functions and relations
  - global constraints
  - alldifferent
  - cumulative

Example

1. Constraint max(X,Y,Z): $(X \geq Y \wedge Z=X) \vee (Y \geq X \wedge Z=Y)$
   Corresponding CLP program

```
max(X,Y,Z) :-   X #>=Z,   Z#=X.
max(X,Y,Z) :-   Y #>= X,  Z #= Y.
```

2. Constraint: "X is an even number", $\exists Y.(X=2 \times Y)$
   Corresponding CLP program: even(X) :- X#=2*Y.
   Not representable extensionally
   Use of local variables which do not need an initial domain

CLP formulation of CSPs

- Allows for the use of local variables
- Allows encapsulation of a CSP as a constraint and making any of its variables local
- Building complex CSPs from simple ones
- Recursive definition of constraints

Important features of CLP

- CLP allows for local variables and recursive definition
  - can express problem with unbounded number of variables
- Representing solutions without fixing all the variables
  - interactive problem solving
  - partial solution observation during search

Important features of CLP(2)

- Allows the programmer to define search strategies
  - expressing the design model
  - backtracking (inherited from LP) combined with reflection predicates
- Allows the programmer to (partially) control how the underlying constraint solver works
  - disjunction
  - reification
  - indexicals
  - constraint handling rules
  - generalized propagation

CLP for design modeling

- In CLP languages
- constraints generated dynamically
- satisfaction tests are performed at intermediate stages
- such tests influence future execution and constraint generation
- → incremental solvers
- solver current state: constraints encountered so far during the derivation
- new constraint added → revise current state and test its satisfiability
- if unsatifiability is detected → return to the last state with unexplored child states (state recovery)
- nothing new: it's backtracking!

Incremental solvers

- Prolog II → incremental solver for equations and disequations
- CLP(R) → incremental simplex
- CHIP → Backtracking + AI techniques (propagation)

Bibliography for CP

- **Handbook of Constraint programming, Rossi, van Beek and Walsh editors, Elsevier 2006.**
- K. R. Apt, Principles of Constraint Programming, Cambridge University Press, 2003.
- R. Dechter, Constraint processing, Morgan Kaufmann, 2003
- A. Mackworth, Consistency in networks of relations, Artificial Intelligence, 8, 1, 1977.
- A. Mackworth, E. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, Artificial Intelligence, 25, 1985.
- U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, Information Science, 7, 66, 1974

Bibliography for CLP

main references for this tutorial

Other important references

- Constraint Logic Programming: Chapter 12 of handbook of Constraint programming, Rossi, van Beek and Walsh editors, Elsevier 2006.
- J. Jaffar, M. J. Maher, K. Marriott and P. J. Stuckey, The Semantics of Constraint Logic Programs, Journal of Logic Programming, volume 37, number 1-3, pages 1-46, 1998.
- K.R. Apt, M. H. van Emden: Contributions to the Theory of Logic Programming. J. ACM 29(3): 841-862 (1982)
- A. M. Cheadle W. Harvey, A. J. Sadler, J. Schimpf K. Shen M. G. Wallace, ECLiPSe: An Introduction, Tech. Report IC-PARC 03 1
- Alain Colmerauer: An Introduction to Prolog III. Commun. ACM 33(7): 69-90 (1990)
- M. Dincbas, P. Van Hentenryck, Helmut Simonis, A. Aggoun, T. Graf, F. BerthierThe Constraint Logic Programming Language CHIP. FGCS 1988: 693-702
- T. W. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, M.Wallace: Contraint Logic Programming - An Informal Introduction. Logic Programming Summer School 1992: 3-35

Bibliography for CLP

Other important references ctd.

- M. Gabbrielli, G. Levi: Modeling Answer Constraints in Constraint Logic Programs. ICLP 1991: 238-252
- J. Jaffar, J-L. Lassez: Constraint Logic Programming. POPL 1987: 111-119.
- J.Jaffar, M. J. Maher: Constraint Logic Programming: A Survey. J. Log. Program. 19/20: 503-581 (1994)
- J. Jaffar, S.Michaylov, P. J. Stuckey, R. H. C. Yap: The CLP(R) Language and System. ACM Trans. Program. Lang. Syst. 14(3): 339-395 (1992)
- K. Marriott, P. J. Stuckey: Semantics of Constraint Logic Programs with Optimization. LOPLAS 2(1-4): 197-212, 1993.
- F. Rossi: Constraint (Logic) Programming: A Survey on Research and Applications. New Trends in Constraints 1999: 40-74
- H. Simonis Tutorial on constraint logic programming. On the web.
- P.Van Hentenryck, H. Simonis, M. Dincbas: Constraint Satisfaction Using Constraint Logic Programming. Artif. Intell. 58(1-3): 113-159 (1992)