Computational Logic

Constraint Logic Programming
Constraints

- Constraint: some form of restriction that a solution must satisfy
  - $X + Y = 20$
  - $X \land Y$ is true
  - The third field of the data structure is greater than the second
  - The murderer is one of those who had met the cabaret entertainer

- CLP: LP plus the ability to compute with some form of constraints (which are being solved by the system during computation)

- Features in CLP:
  - Domain of computation (reals, rationals, integers, booleans, structures, etc.)
  - Type of expressions on a domain ($+, *, \land, \lor$)
  - Type of constraints allowed: equations, disequations, inequations, etc. ($=, \neq, \leq, \geq, <, >$)
  - Constraint solving algorithms: simplex, gauss, etc.
A Comparison with LP (I)

- Example (Prolog): \( q(X, Y, Z) :- Z = f(X, Y) \).
  \[
  \begin{align*}
  &\mid \ ?- \ q(3, 4, Z).
  \quad Z = f(3,4) \\
  &\mid \ ?- \ q(X, Y, f(3,4)).
  \quad X = 3, Y = 4 \\
  &\mid \ ?- \ q(X, Y, Z).
  \quad Z = f(X,Y)
  \end{align*}
  \]

- Example (Prolog): \( p(X, Y, Z) :- Z \text{ is } X + Y \).
  \[
  \begin{align*}
  &\mid \ ?- \ p(3, 4, Z).
  \quad Z = 7 \\
  &\mid \ ?- \ p(X, 4, 7).
  \quad \text{INSTANTIATION ERROR: in expression}
  \end{align*}
  \]
A Comparison with LP (II)

- Example (CLP): \( p(X, Y, Z) :- Z = X + Y. \)

  2 \(?-\) \( p(3, 4, Z). \)
  
  \[ Z = 7 \]
  
  *** Yes

  3 \(?-\) \( p(X, 4, 7). \)
  
  \[ X = 3 \]
  
  *** Yes

  4 \(?-\) \( p(X, Y, 7). \)
  
  \[ X = 7 - Y \]
  
  *** Yes
A Comparison with LP (III)

- Advantages:
  - Helps making programs expressive and flexible.
  - May save much coding.
  - In some cases, more efficient than traditional LP programs due to solvers typically being very efficiently implemented.
  - Also, efficiency due to search space reduction:
    * LP: generate-and-test.
    * CLP: constrain-and-generate.

- Disadvantages:
  - Complexity of solver algorithms (simplex, gauss, etc) can affect performance.

- Solutions:
  - better algorithms
  - compile-time optimizations (program transformation, global analysis, etc)
  - parallelism
Example of Search Space Reduction

- Prolog (generate–and–test):
  solution(X, Y, Z) :-
    p(X), p(Y), p(Z),
    test(X, Y, Z).


  test(X, Y, Z) :- Y is X + 1, Z is Y + 1.

- Query:

  | ?- solution(X, Y, Z).
  X = 14
  Y = 15
  Z = 16 ? ;
  no

- 458 steps (all solutions: 475 steps).
Example of Search Space Reduction

- **CLP (generate–and–test):**
  
  solution(X, Y, Z) :-
  
  p(X), p(Y), p(Z),
  
  test(X, Y, Z).
  

  test(X, Y, Z) :- Y = X + 1, Z = Y + 1.

- **Query:**
  
  ?- solution(X, Y, Z).
  
  Z = 16
  
  Y = 15
  
  X = 14
  
  *** Retry? y
  
  *** No
  
- **458 steps (all solutions: 475 steps).**
Generate–and–test Search Tree

A5

Y=14 Y=15

A4 A3 A2 A1

X=15 X=16 X=7 X=3 X=11 X=14

Z=14 Z=15 Z=16 Z=7 Z=3 Z=11

B5

B4 B3 B2 B1

Y=14 Y=15 Y=16 Y=7 Y=3 Y=11

Z=15 Z=16 Z=7 Z=3 Z=11

B

A
Example of Search Space Reduction

- **Move** \texttt{test(X, Y, Z)} at the beginning (constrain–and–generate):
  ```prolog
  solution(X, Y, Z) :-
      test(X, Y, Z),
      p(X), p(Y), p(Z).
  ```

- **Prolog**: \texttt{test(X, Y, Z) :- Y is X + 1, Z is Y + 1.}
  ```prolog
  | ?- solution(X, Y, Z).{INSTANTIATION ERROR: in expression}
  ```

- **CLP**: \texttt{test(X, Y, Z) :- Y = X + 1, Z = Y + 1.}
  ```prolog
  ?- solution(X, Y, Z).
  Z = 16
  Y = 15
  X = 14
  *** Retry? y
  *** No
  ```

- 11 steps (all solutions: 11 steps).
Constrain–and–generate Search Tree
Constraint Systems: CLP(𝒳)

- Semantics parameterized by the constraint domain: CLP(𝒳), where 𝒳 ≡ (Σ, D, ℒ, 𝒯)
- Signature Σ: set of predicate and function symbols, together with their arity
- ℒ ⊆ Σ–formulae: constraints
- D is the set of actual elements in the domain
- Σ–structure D: gives the meaning of predicate and function symbols (and hence, constraints).
- 𝒯 a first–order theory (axiomatizes some properties of D)
- (D, ℒ) is a constraint domain
- Assumptions:
  - ℒ built upon a first–order language
  - ∈ Σ is identity in D
  - There are identically false and identically true constraints in ℒ
  - ℒ is closed w.r.t. renaming, conjunction and existential quantification
Constraint Domains (I)

- \( \Sigma = \{0, 1, +, *, =, <, \leq\} \), \( D = \mathbb{R} \), \( D \) interprets \( \Sigma \) as usual, \( R = (D, L) \)
  
  ◦ Arithmetic over the reals
  
  ◦ Eg.: \( x^2 + 2xy < \frac{y}{x} \land x > 0 \) \( \equiv xxx + xxy + xxy < y \land 0 < x \)

- Question: is 0 needed? How can it be represented?

- Let us assume \( \Sigma' = \{0, 1, +, =, <, \leq\} \), \( R_{Lin} = (D', L') \)
  
  ◦ Linear arithmetic
  
  ◦ Eg.: \( 3x - y < 3 \) \( \equiv x + x + x < 1 + 1 + 1 + y \)

- Let us assume \( \Sigma'' = \{0, 1, +, =\} \), \( R_{LinEq} = (D'', L'') \)
  
  ◦ Linear equations
  
  ◦ Eg.: \( 3x + y = 5 \land y = 2x \)
Constraint Domains (II)

- $\Sigma = \{ <\text{constant and function symbols}, = \} \}
- D = \{ \text{finite trees} \}
- $D$ interprets $\Sigma$ as tree constructors
- Each $f \in \Sigma$ with arity $n$ maps $n$ trees to a tree with root labeled $f$ and whose subtrees are the arguments of the mapping
- Constraints: syntactic tree equality
- $\mathcal{FT} = (D, L)$
  - Constraints over the Herbrand domain
  - Eg.: $g(h(Z), Y) = g(Y, h(a))$
- LP $\equiv$ CLP($\mathcal{FT}$)
- LP can be viewed as a constraint logic language over Herbrand terms with a single constraint predicate symbol: “$=$”
Constraint Domains (III)

- \( \Sigma = \{ <\text{constants}>, \lambda, ., ::, = \} \)
- \( D = \{ \text{finite strings of constants} \} \)
- \( D \) interprets . as string concatenation, :: as string length
  - Equations over strings of constants
  - Eg.: \( X.A.X = X.A \)

- \( \Sigma = \{ 0, 1, \neg, \land, = \} \)
- \( D = \{ \text{true}, \text{false} \} \)
- \( D \) interprets symbols in \( \Sigma \) as boolean functions
- \( \text{BOOL} = (D, L) \)
  - Boolean constraints
  - Eg.: \( \neg(x \land y) = 1 \)
CLP(\mathcal{L}') Programs

- Recall that:
  - $\Sigma$ is a set of predicate and function symbols
  - $\mathcal{L} \subseteq \Sigma$–formulae are the constraints
- $\Pi$: set of predicate symbols definable by a program
- Atom: $p(t_1, t_2, \ldots, t_n)$, where $t_1, t_2, \ldots, t_n$ are terms and $p \in \Pi$
- Primitive constraint: $p(t_1, t_2, \ldots, t_n)$, where $t_1, t_2, \ldots, t_n$ are terms and $p \in \Sigma$ is a predicate symbol
- Every constraint is a (first–order) formula built from primitive constraints
- The class of constraints will vary (generally only a subset of formulas are considered constraints)
- A CLP program is a collection of rules of the form $a \leftarrow b_1, \ldots, b_n$ where $a$ is an atom and the $b_i$’s are atoms or constraints
- A fact is a rule $a \leftarrow c$ where $c$ is a constraint
- A goal (or query) $G$ is a conjunction of constraints and atoms
Issues in CLP

- CLP may use the same execution strategy as Prolog (depth-first, left-to-right) or a different one.
- Prolog arithmetics (i.e., is/2) may remain or simply disappear, substituted by constraint solving.
- Syntax may vary upon systems:
  ◦ Different constraint systems use different symbols for constraints:
    * = for unification, #=, .=., etc. for constraints
  ◦ Overloading: equations are subsumed by /=2 (extended unification)
    * A=f(X,Y) is regarded as unification
    * A=X+Y is regarded as a constraint
- Head unification may remain as plain or extended unification:
  Call ?- p(A) with clause head p(X+Y) :- yields equation A=X+Y
  ◦ a unification equation
  ◦ a constraint
CLP(ℜ): A case study

- Arithmetics over the reals

- For the examples we assume:
  - Same execution strategy as Prolog
  - Equations and disequations are allowed
  - Linear constraints are solved, non–linear constraints are passive: delayed until linear or simple checks
    * $X \cdot Y = 7$ becomes linear when $X$ is assigned a single value
    * $X \cdot X + 2 \cdot X + 1 = 0$ becomes a check when $X$ is assigned a single value
  - Prolog arithmetics disappears, subsumed by constraint solving
  - Overloading and extended unification is used
  - Head unification is extended for constraint solving
Linear Equations (CLP(ℜ))

- Vector × vector multiplication (dot product):
  \[ (x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = x_1 \cdot y_1 + \cdots + x_n \cdot y_n \]

- Vectors represented as lists of numbers
  ```prolog
  prod([], [], 0).
  prod([X|Xs], [Y|Ys], X * Y + Rest) :-
      prod(Xs, Ys, Rest).
  ```

- Unification becomes constraint solving!
  ```prolog
  ?- prod([2, 3], [4, 5], K).
  K = 23
  ?- prod([2, 3], [5, X2], 22).
  X2 = 4
  ?- prod([2, 7, 3], [Vx, Vy, Vz], 0).
  Vx = -1.5*Vz - 3.5*Vy
  ```

- Any computed answer is, in general, an equation over the variables in the query
Systems of Linear Equations (CLP(ℜ))

- Can we solve systems of equations? E.g.,
  \[
  3x + y = 5 \\
  x + 8y = 3
  \]

- Write them down at the top level prompt:
  \[
  \text{?- prod([3, 1], [X, Y], 5), prod([1, 8], [X, Y], 3).} \\
  X = 1.6087, \ Y = 0.173913
  \]

- A more general predicate can be built mimicking the mathematical vector notation
  \[
  \mathbf{A} \cdot \mathbf{x} = \mathbf{b}:
  \]

  \[
  \text{system(_Vars, [], []).} \\
  \text{system(Vars, [Co|Coefs], [Ind|Indeps]) :-} \\
  \quad \text{prod(Vars, Co, Ind),} \\
  \quad \text{system(Vars, Coefs, Indeps).}
  \]

- We can now express (and solve) equation systems
  \[
  \text{?- system([X, Y], [[3, 1], [1, 8]], [5, 3]).} \\
  X = 1.6087, \ Y = 0.173913
  \]
Non–linear Equations (CLP(ℜ))

- Non–linear equations are delayed
  ?- \( \sin(X) = \cos(X) \).  
  \( \sin(X) = \cos(X) \)

- This is also the case if there exists some procedure to solve them
  ?- \( X \cdot X + 2 \cdot X + 1 = 0 \).  
  -2 \cdot X - 1 = X \cdot X

- Reason: no general solving technique is known. CLP(ℜ) solves only linear (dis)equations.

- Once equations become linear, they are handled properly:
  ?- \( X = \cos(\sin(Y)) \), \( Y = 2 + Y \cdot 3 \).  
  \( Y = -1 \), \( X = 0.666367 \)

- Disequations are solved using a modified, incremental Simplex
  ?- \( X + Y \leq 4 \), \( Y \geq 4 \), \( X \geq 0 \).  
  \( Y = 4 \), \( X = 0 \)
Fibonacci Revisited (Prolog)

- Fibonacci numbers:
  \[ F_0 = 0 \]
  \[ F_1 = 1 \]
  \[ F_{n+2} = F_{n+1} + F_n \]

- (The good old) Prolog version:
  \[
  \begin{align*}
  \text{fib}(0, 0). \\
  \text{fib}(1, 1). \\
  \text{fib}(N, F) :- \\
  \quad N > 1, \\
  \quad N1 \text{ is } N - 1, \\
  \quad N2 \text{ is } N - 2, \\
  \quad \text{fib}(N1, F1), \\
  \quad \text{fib}(N2, F2), \\
  \quad F \text{ is } F1 + F2.
  \end{align*}
  \]

- Can only be used with the first argument instantiated to a number
Fibonacci Revisited (CLP(ℜ))

- CLP(ℜ) version: syntactically similar to the previous one

  fib(0, 0).
  fib(1, 1).
  fib(N, F1 + F2) :-
      N > 1, F1 >= 0, F2 >= 0,
      fib(N - 1, F1), fib(N - 2, F2).

- Note all constraints included in program (F1 >= 0, F2 >= 0) – good practice!

- Only real numbers and equations used (no data structures, no other constraint system): “pure CLP(ℜ)”

- Semantics greatly enhanced! E.g.

  ?- fib(N, F).
  F = 0, N = 0 ;
  F = 1, N = 1 ;
  F = 1, N = 2 ;
  F = 2, N = 3 ;
  F = 3, N = 4 ;
Analog RLC circuits (CLP(ℝ))

- Analysis and synthesis of analog circuits
- RLC network in steady state
- Each circuit is composed either of:
  - A simple component, or
  - A connection of simpler circuits
- For simplicity, we will suppose subnetworks connected only in parallel and series → Ohm’s laws will suffice (other networks need global, i.e., Kirchoff’s laws)
- We want to relate the current (I), voltage (V) and frequency (W) in steady state
- Entry point: circuit(C, V, I, W) states that:
  - across the network C, the voltage is V, the current is I and the frequency is W
- V and I must be modeled as complex numbers (the imaginary part takes into account the angular frequency)
- Note that Herbrand terms are used to provide data structures
Analog RLC circuits (CLP(ℜ))

- Complex number $X + Yi$ modeled as $c(X, Y)$
- Basic operations:

  \[
  \begin{align*}
  \text{c_add} & (c(Re_1, Im_1), c(Re_2, Im_2), c(Re_1+Re_2, Im_1+Im_2)). \\
  \text{c_mult} & (c(Re_1, Im_1), c(Re_2, Im_2), c(Re_3, Im_3)) :- \\
  & \text{Re}\_3 = Re_1 \times Re_2 - Im_1 \times Im_2, \\
  & \text{Im}\_3 = Re_1 \times Im_2 + Re_2 \times Im_1.
  \end{align*}
  \]

  (equality is $c\_\text{equal}(c(R, I), c(R, I))$, can be left to [extended] unification)
Analog RLC circuits (CLP(ℜ))

• Circuits in series:

\[
\text{circuit(series(N1, N2), V, I, W) :-}
\]
\[
c\_add(V1, V2, V),
\]
\[
circuit(N1, V1, I, W),
\]
\[
circuit(N2, V2, I, W).
\]

• Circuits in parallel:

\[
\text{circuit(parallel(N1, N2), V, I, W) :-}
\]
\[
c\_add(I1, I2, I),
\]
\[
circuit(N1, V, I1, W),
\]
\[
circuit(N2, V, I2, W).
\]
Analog RLC circuits (CLP(ℜ))

Each basic component can be modeled as a separate unit:

- **Resistor**: \( V = I \cdot (R + 0i) \)

  \[ \text{circuit(resistor}(R), V, I, _W) :- \]
  \[ \text{c_mult}(I, c(R, 0), V). \]

- **Inductor**: \( V = I \cdot (0 + WL_i) \)

  \[ \text{circuit(inductor}(L), V, I, W) :- \]
  \[ \text{c_mult}(I, c(0, W \cdot L), V). \]

- **Capacitor**: \( V = I \cdot (0 - \frac{1}{WC}i) \)

  \[ \text{circuit(capacitor}(C), V, I, W) :- \]
  \[ \text{c_mult}(I, c(0, -1 / (W \cdot C)), V). \]
Analog RLC circuits (CLP(ℜ))

- Example:

\[
\begin{align*}
R &= ? \\
C &= ? \\
L &= 0.073 \\
V &= 4.5 \\
I &= 0.65 \\
\omega &= 2400
\end{align*}
\]

?- circuit(parallel(inductor(0.073),
series(capacitor(C), resistor(R))),
c(4.5, 0), c(0.65, 0), 2400).

\[
R = 6.91229, C = 0.00152546
\]

?- circuit(C, c(4.5, 0), c(0.65, 0), 2400).
The N Queens Problem

- Problem:
  place \( N \) chess queens in a \( N \times N \) board such that they do not attack each other
- Data structure: a list holding the column position for each row
- The final solution is a permutation of the list \([1, 2, \ldots, N]\)
- E.g.: the solution is represented as \([2, 4, 1, 3]\)
- General idea:
  ◦ Start with partial solution
  ◦ Nondeterministically select new queen
  ◦ Check safety of new queen against those already placed
  ◦ Add new queen to partial solution if compatible; start again with new partial solution
The N Queens Problem (Prolog)

queens(N, Qs) :- queens_list(N, Ns), queens(Ns, [], Qs).

queens([], Qs, Qs).

queens(Unplaced, Placed, Qs) :-
    select(Unplaced, Q, NewUnplaced), no_attack(Placed, Q, 1),
    queens(NewUnplaced, [Q|Placed], Qs).

no_attack([], _Queen, _Nb).

no_attack([Y|Ys], Queen, Nb) :-
    Queen =\= Y + Nb, Queen =\= Y - Nb, Nb1 is Nb + 1,
    no_attack(Ys, Queen, Nb1).

select([X|Ys], X, Ys).

select([Y|Ys], X, [Y|Zs]) :- select(Ys, X, Zs).

queens_list(0, []).

queens_list(N, [N|Ns]) :- N > 0, N1 is N - 1, queens_list(N1, Ns).
The N Queens Problem (Prolog)
The N Queens Problem (CLP(\mathcal{R}))

queens(N, Qs) :- constrain_values(N, N, Qs), place_queens(N, Qs).

constrain_values(0, _N, []).
constrain_values(N, Range, [X|Xs]) :-
    N > 0, X > 0, X <= Range,
    constrain_values(N - 1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
    abs(Queen - (Y + Nb)) > 0, % Queen =\= Y + Nb
    abs(Queen - (Y - Nb)) > 0, % Queen =\= Y - Nb
    no_attack(Ys, Queen, Nb + 1).

place_queens(0, _).
place_queens(N, Q) :- N > 0, member(N, Q), place_queens(N - 1, Q).

member(X, [X|_]).
member(X, [_|Xs]) :- member(X, Xs).
The N Queens Problem (CLP(ℜ))

- This last program can attack the problem in its most general instance:

  ```prolog
  ?- queens(M,N).
  N = [], M = 0 ;
  M = [1], M = 1 ;
  N = [2, 4, 1, 3], M = 4 ;
  N = [3, 1, 4, 2], M = 4 ;
  N = [5, 2, 4, 1, 3], M = 5 ;
  N = [5, 3, 1, 4, 2], M = 5 ;
  N = [3, 5, 2, 4, 1], M = 5 ;
  N = [2, 5, 3, 1, 4], M = 5
  ...
  
  Remark: Herbrand terms used to build the data structures
  
  But also used as constraints (e.g., length of already built list Xs in no_attack(Xs, X, 1))
  
  Note that in fact we are using both ℜ and ℱȚ
The N Queens Problem (CLP(ℜ))
The N Queens Problem (CLP(ℜ))

- CLP(ℜ) generates internally a set of equations for each board size
- They are non-linear and are thus delayed until instantiation wakes them up

?- constrain_values(4, 4, Q).
Q = [_t3, _t5, _t13, _t21]
_t3 <= 4 0 < abs(-_t13 + _t3 - 2)
_t5 <= 4 0 < abs(-_t13 + _t3 + 2)
_t13 <= 4 0 < abs(-_t21 + _t3 - 3)
_t21 <= 4 0 < abs(-_t21 + _t3 + 3)
0 < _t3 0 < abs(-_t13 + _t5 - 1)
0 < _t5 0 < abs(-_t13 + _t5 + 1)
0 < _t13 0 < abs(-_t21 + _t5 - 2)
0 < _t21 0 < abs(-_t21 + _t5 + 2)
0 < abs(-_t5 + _t3 - 1) 0 < abs(-_t21 + _t13 - 1)
0 < abs(-_t5 + _t3 + 1) 0 < abs(-_t21 + _t13 + 1)
The N Queens Problem (CLP(ℜ))

• Constraints are (incrementally) simplified as new queens are added

?- constrain_values(4, 4, Qs), Qs = [3,1|0Qs].

0Qs = [_t16, _t24] 0 < abs(-_t24)
Qs = [3, 1, _t16, _t24] 0 < abs(-_t24 + 6)
_t16 <= 4 0 < abs(-_t16)
_t24 <= 4 0 < abs(-_t16 + 2)
0 < _t16 0 < abs(-_t24 - 1)
0 < _t24 0 < abs(-_t24 + 3)
0 < abs(-_t16 + 1) 0 < abs(-_t24 + _t16 - 1)
0 < abs(-_t16 + 5) 0 < abs(-_t24 + _t16 + 1)

• Bad choices are rejected using constraint consistency:

?- constrain_values(4, 4, Qs), Qs = [3,2|0Qs].

*** No
CLP($\mathcal{FD}$): Finite Domains

- Arithmetics over integers

- A *finite domain* constraint solver associates each variable with a finite subset of $\mathbb{Z}$

- Example: $E \in \{-123, -10..4, 10\}$
  - $E :: [-123, -10..4, 10]$ (Eclipse notation)
  - $E$ in $\{-123\} \setminus (-10..4) \setminus \{10\}$ (SICStus notation)
  - *We will use* $E$ in $[-123, -10..4, 10]$
    (without list construct if the list is a singleton)
Finite Domains (I)

- We can:
  - Establish the *domain* of a variable (*in*)
  - Perform arithmetic operations (+, −, *, /) on the variables
  - Establish linear relationships among arithmetic expressions (# =, # <, # =<)

- Those operations / relationships are intended to narrow the domains of the variables

- Note:
  - SICStus requires the use in the source code of the directive
    ```prolog
    :- use_module(library(clpfd)).
    ```
  - Ciao requires the use of
    ```prolog
    :- use_package(fd).
    ```
Finite Domains (II)

• Example:

  \[ ?- X \neq A + B, \text{ A in 1..3, B in 3..7.} \]
  \[ X \text{ in 4..10, A in 1..3, B in 3..7} \]

• The respective minimums and maximums are added

• There is no unique solution

  \[ ?- X \neq A - B, \text{ A in 1..3, B in 3..7.} \]
  \[ X \text{ in } -6..0, A \text{ in 1..3, B in 3..7} \]

• The minimum value of \( X \) is the minimum value of \( A \) minus the maximum value of \( B \)

• (Similar for the maximum values)

• Putting more constraints:

  \[ ?- X \neq A - B, \text{ A in 1..3, B in 3..7, } X \geq 0. \]
  \[ A = 3, B = 3, X = 0 \]
Finite Domains (III)

Some useful primitives in finite domains:

- `fd_min(X, T)`: the term T is the minimum value in the domain of the variable X
  - This can be used to minimize (c.f., maximize) a solution
    ```prolog
    ?- X #= A - B, A in 1..3, B in 3..7, fd_min(X, X).
    A = 1, B = 7, X = -6
    ```

- `domain(Variables, Min, Max)`: A shorthand for several in constraints

- `labeling(Options, VarList)`: 
  - instantiates variables in VarList to values in their domains
  - Options dictates the search order

  ```prolog
  ?- X*X+Y*Y#=Z*Z, X#>=Y, domain([X, Y, Z],1,1000),labeling([],[X,Y,Z]).
  X = 4, Y = 3, Z = 5
  X = 8, Y = 6, Z = 10
  X = 12, Y = 5, Z = 13
  ...```
A Project Management Problem (I)

- The job whose dependencies and task lengths are given by: should be finished in 10 time units or less

- Constraints:

  \[
  \text{pn1}(A, B, C, D, E, F, G) :-
  \begin{align*}
  &A \geq 0, \ G \leq 10, \\
  &B \geq A, \ C \geq A, \ D \geq A, \\
  &E \geq B + 1, \ E \geq C + 2, \\
  &F \geq C + 2, \ F \geq D + 3, \\
  &G \geq E + 4, \ G \geq F + 1.
  \end{align*}
  \]
A Project Management Problem (II)

- Query:

  ```prolog
  ?- pn1(A,B,C,D,E,F,G).
  A in 0..4, B in 0..5, C in 0..4,
  D in 0..6, E in 2..6, F in 3..9, G in 6..10,
  ```

- Note the slack of the variables

- Some additional constraints must be respected as well, but are not shown by default

- Minimize the total project time:

  ```prolog
  ?- pn1(A,B,C,D,E,F,G), fd_min(G, G).
  A = 0, B in 0..1, C = 0, D in 0..2,
  E = 2, F in 3..5, G = 6
  ```

- Variables without slack represent critical tasks
A Project Management Problem (III)

- An alternative setting:

- We can accelerate task $F$ at some cost

  $$pn2(A, B, C, D, E, F, G, X) :-$$
  $$A \#>\!\!=\!\!= 0, G \#=< 10,$$
  $$B \#>= A, C \#>= A, D \#>= A,$$
  $$E \#>= B + 1, E \#>= C + 2,$$
  $$F \#>= C + 2, F \#>= D + 3,$$
  $$G \#>= E + 4, G \#>= F + X.$$

- We do not want to accelerate it more than needed!

  $$?- \ pn2(A, B, C, D, E, F, G, X),$$
  $$\text{fd\_min}(G, G), \text{fd\_max}(X, X).$$
  $$A = 0, B \ in \ 0..1, C = 0, D = 0,$$
  $$E = 2, F = 3, G = 6, X = 3$$
A Project Management Problem (IV)

• We have two independent tasks B and D whose lengths are not fixed:

• We can finish any of B, D in 2 time units at best

• Some shared resource disallows finishing both tasks in 2 time units: they will take 6 time units
A Project Management Problem (V)

- Constraints describing the net:

\[
\text{pn3}(A,B,C,D,E,F,G,X,Y) :- \\
A #>= 0, G #=< 10, \\
X #>= 2, Y #>= 2, X + Y #= 6, \\
B #>= A, C #>= A, D #>= A, \\
E #>= B + X, E #>= C + 2, \\
F #>= C + 2, F #>= D + Y, \\
G #>= E + 4, G #>= F + 1.
\]

- Query: 
  \[- \text{pn3}(A,B,C,D,E,F,G,X,Y), \text{fd\_min}(G,G). \]
  
  \( A=0, B=0, C=0, D \text{ in } 0..1, E=2, F \text{ in } 4..5, X=2, Y=4, G=6 \)

- I.e., we must devote more resources to task \( B \)
- All tasks but \( F \) and \( D \) are critical now
- Sometimes, \( \text{fd\_min}/2 \) not enough to provide best solution (pending constraints):
  
  \[- \text{pn3}(A,B,C,D,E,F,G,X,Y), \\
  \text{labeling([[ff, minimize(G)]}, [A,B,C,D,E,F,G,X,Y])}. \]
The N-Queens Problem Using Finite Domains (in SICStus Prolog)

- By far, the fastest implementation
  queens(N, Qs, Type) :-
    constrain_values(N, N, Qs),
    all_different(Qs), % built-in constraint
    labeling(Type, Qs).

  constrain_values(0, _N, []).
  constrain_values(N, Range, [X|Xs]) :-
    N > 0, N1 is N - 1, X in 1 .. Range,
    constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

  no_attack([], _Queen, _Nb).
  no_attack([Y|Ys], Queen, Nb) :-
    Queen #\= Y + Nb, Queen #\= Y - Nb, Nb1 is Nb + 1,
    no_attack(Ys, Queen, Nb1).

- Query. Type is the type of search desired.
  ?- queens(20, Q, [ff]).
  Q = [1,3,5,14,17,4,16,7,12,18,15,19,6,10,20,11,8,2,13,9]?
CLP(\(\mathcal{WE}\))

- Equations over finite strings
- Primitive constraints: concatenation (\(\cdot\)), string length (\(::\))
- Find strings meeting some property:

  ?- "123".z = z."231", z::0.  
  no

  ?- "123".z = z."231", z::1.  
  z = "1"

  ?- "123".z = z."231", z::2.  
  no

  ?- "123".z = z."231", z::3.  
  no

  ?- "123".z = z."231", z::4.  
  z = "1231"

- These constraint solvers are very complex
- Often incomplete algorithms are used
Word equations plus arithmetic over $\mathbb{Q}$ (rational numbers)

Prove that the sequence $x_{i+2} = |x_{i+1}|-x_i$ has a period of length 9 (for any starting $x_0, x_1$)

Strategy: describe the sequence, try to find a subsequence such that the period condition is violated

Sequence description (syntax is Prolog III slightly modified):

```
seq(<Y, X>). abs(Y, Y) :- Y >= 0.
seq(<Y1 - X, Y, X>.U) :- abs(Y, -Y) :- Y < 0.
    seq(<Y, X>.U)
    abs(Y, Y1).
```

Query: Is there any 11-element sequence such that the 2-tuple initial seed is different from the 2-tuple final subsequence (the seed of the rest of the sequence)?

```
?- seq(U.V.W), U::2, V::7, W::2, U#W.
fail
```
CLP(\mathcal{FT}) (a.k.a. Logic Programming)

- Equations over Finite Trees
- Check that two trees are isomorphic (same elements in each level)

```prolog
iso(Tree, Tree).
iso(t(R, I1, D1), t(R, I2, D2)) :-
    iso(I1, D2),
    iso(D1, I2).

?- iso(t(a, b, t(X, Y, Z)), t(a, t(u, v, W), L)).
L=b, X=u, Y=v, Z=W ? ;
L=b, X=u, Y=W, Z=v ? ;
L=b, W=t(_C,_B,_A), X=u, Y=t(_C,_A,_B), Z=v ? ;
L=b, W=t(_E,t(_D,_C,_B),_A), X=u, Y=t(_E,_A,t(_D,_B,_C)), Z=v ?
```

48
Summarizing

• **In general:**
  ◦ Data structures (Herbrand terms) for free
  ◦ Each logical variable may have constraints associated with it (and with other variables)

• **Problem modeling:**
  ◦ Rules represent the problem at a high level
    * Program structure, modularity
    * Recursion used to set up constraints
  ◦ Constraints encode problem conditions
  ◦ Solutions also expressed as constraints

• **Combinatorial search problems:**
  ◦ CLP languages provide backtracking: enumeration is easy
  ◦ Constraints keep the search space manageable

• **Tackling a problem:**
  ◦ Keep an open mind: often new approaches possible
Complex Constraints

- Some complex constraints allow expressing simpler constraints
- May be operationally treated as passive constraints
- E.g.: cardinality operator \( \#(L, [c_1, \ldots, c_n], U) \) meaning that the number of true constraints lies between \( L \) and \( U \) (which can be variables themselves)
  - If \( L = U = n \), all constraints must hold
  - If \( L = U = 1 \), one and only one constraint must be true
  - Constraining \( U = 0 \), we force the conjunction of the negations to be true
  - Constraining \( L > 0 \), the disjunction of the constraints is specified

- Disjunctive constructive constraint: \( c_1 \lor c_2 \)
  - If properly handled, avoids search and backtracking

- E.g.: 
  \[
  \text{nz}(X) \leftarrow X > 0. \\
  \text{nz}(X) \leftarrow X < 0. \\
  \text{nz}(X) \leftarrow X < 0 \lor X > 0.
  \]
Other Primitives

• CLP(\(\mathcal{X}\)) systems usually provide additional primitives

• E.g.:
  ◦ \texttt{enum}(X) enumerates \(X\) inside its current domain
  ◦ \texttt{maximize}(X)\ (c.f. \texttt{minimize}(X)) works out maximum (minimum value) for \(X\) under the active constraints
  ◦ \texttt{delay Goal until Condition} specifies when the variables are instantiated enough so that \texttt{Goal} can be effectively executed
    * Its use needs deep knowledge of the constraint system
    * Also widely available in Prolog systems
    * Not really a constraint: control primitive
Programming Tips

• Over-constraining:
  ◦ Seems to be against general advice “do not perform extra work”, but can actually cut more space search
  ◦ Specially useful if infer is weak
  ◦ Or else, if constraints outside the domain are being used

• Use control primitives (e.g., cut) very sparingly and carefully

• Determinacy is more subtle, (partially due to constraints in non–solved form)

• Choosing a clause does not preclude trying other exclusive clauses (as with Prolog and plain unification)

• Compare:

  `{max(X,Y,Z) :- X > Y. Z = X, Y < X ;
   max(X,Y,Y) :- X <= Y. Z = Y, X <= Y
   max(X,Y,X) :- X > Y, !. Z = X, Y < X ;
   max(X,Y,Y) :- X <= Y. no
`
Some Real Systems (I)

- CLP defines a class of languages obtained by
  ◦ Specifying the particular constraint system(s)
  ◦ Specifying *Computation* and *Selection* rules

- Most share the Herbrand domain with “=”, but add different domains and/or solver algorithms

- Most use *Computation* and *Selection* rules of Prolog

- CLP(ℜ):
  ◦ Linear arithmetic over reals (\(=, \leq, >\))
  ◦ Gauss elimination and an adaptation of Simplex

- PrologIII:
  ◦ Linear arithmetic over rationals (\(=, \leq, >, \neq\)), Simplex
  ◦ Boolean (\(=\)), 2-valued Boolean Algebra
  ◦ Infinite (rational) trees (\(=, \neq\))
  ◦ Equations over finite strings
Some Real Systems (II)

- **CHIP:**
  - Linear arithmetic over rationals ($=, \leq, >, \neq$), Simplex
  - Boolean ($=$), larger Boolean algebra (symbolic values)
  - Finite domains
  - User–defined constraints and solver algorithms

- **BNR-Prolog:**
  - Arithmetic over reals (closed intervals) ($=, \leq, >, \neq$), Simplex, propagation techniques
  - Boolean ($=$), 2-valued Boolean algebra
  - Finite domains, consistency techniques under user–defined strategy

- **SICStus 3:**
  - Linear arithmetic over reals ($=, \leq, >, \neq$)
  - Linear arithmetic over rationals ($=, \leq, >, \neq$)
  - Finite domains (in recent versions)
Some Real Systems (III)

- **ECL$^i$PS$^e$:**
  - Finite domains
  - Linear arithmetic over reals ($=, \leq, >, \neq$)
  - Linear arithmetic over rationals ($=, \leq, >, \neq$)

- **clp(FD)/gprolog:**
  - Finite domains

- **RISC–CLP:**
  - Real arithmetic terms: any arithmetic constraint over reals
  - Improved version of Tarski’s quantifier elimination

- **Ciao:**
  - Linear arithmetic over reals ($=, \leq, >, \neq$)
  - Linear arithmetic over rationals ($=, \leq, >, \neq$)
  - Finite Domains (currently interpreted)

(can be selected on a per-module basis)