Constraint Logic Programming

Manuel Carro
mcarro@fi.upm.es
Technical University of Madrid (Spain)

What is CLP

Constraints for Problem Solving

- Born within AI: e.g. house design
- Constraints as problem representation

  The man in yellow does not have green eyes
  The murderer knows no detective will ever wear dark clothes
- Need an appropriate constraint system (and solver)
- Solution: assignment which agrees with c initial constraints

  Murderer: López, green eyes, Magnum gun
- Or, alternatively, constraints as problem solution (may express different solutions):

  The murderer is one of those who had met the cabaret entertainer
- There might be no solution: *Natural death*

A General View

- Ancestors: SKETCHPAD (1963), Waltz's algorithm (1965?), THINGLAB (1981), MACSYMA (1983)
- Constraints in logic languages:
  - Idea of *Constraint Programming*
  - General theory
  - In practice: based on Prolog + some constraint domain
- Constraints in imperative languages:
  - Equation solving libraries (ILOG)
  - Timestamping of variables:

    `x := x + 1` $\leftrightarrow x_i := x_{i+1} + 1$
- Constraints in functional languages:
  - Evaluation of expressions with free variables:
  - *Absolute Set Abstraction*

Constraints: High–level Problem Modelling

- Think of constraints as (dis)equations over arbitrary items
- Use constraint system primitives to encode the problem conditions
- But:
  - Lack of modularity
  - Creating constraints dynamically not easy —must be statically defined
  - Probably constraint system not powerful enough to reflect the whole problem
  - Or solutions not given in the desired format

Programming:

- Use programming techniques:
  - Data structures and data abstraction
  - Ad–hoc algorithms, when desired / advantageous
  - Modularity
- Computational power: host language is enriched
- Set up constraints during program execution
- Possibility of adding control:
  - Data flow
  - Program execution
  - Constraint solving
- External communication

Logic

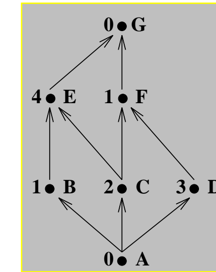
- Declarative reading:
  - Referential transparency
  - Small code units have their own, isolated meaning

  (much like mathematical equations)
- Logical (mathematical) variables:
  - Bi–directional parameter passing
    (more than pattern matching)
  - Single assignment
  - No need for explicit memory management
  - Data structures easy to build
- Non–determinism: built–in search procedure

Why Constraints and Programming?

- Constraints offer a powerful modeling tool
- But in real problems, there are...
  - ...decisions which can be made in either way (and we do not know before the right one) ⟶ search (also: disjunctive constraints)
  - ...weak constraints which may or may not be satisfied
  - ...an idea of the stages a task can be subdivided in
  - ...
- Example: a program to help in house design
- Coupling constraint processing and programming gives more power than any of them separately
- We know about programming; what is the deal with constraints?

Be a Solver (I)



- Assume the precedence net and task lengths on the left
- The whole job must be finished in 10 time units or less
- Problem modelling:

$$
\begin{array}{rcl}
a,b,c,d,e,f,g & \in & \{0,\ldots,10\} \\
a & \leq & b,c,d \\
b+1 & \leq & e \\
c+2 & \leq & e \\
c+2 & \leq & f \\
d+3 & \leq & f \\
e+4 & \leq & g \\
f+1 & \leq & g
\end{array}
$$

Be a Solver (II)

- We will use *finite domain* ($\mathcal{FD}$) constraints:
  - Each variable has a (finite) domain associated i.e., a set of natural numbers
    - `X ∈ {1,2,3,4,7,8,9} ≡ X in 1..4 ∨7..9`
  - There are (dis)equations which relate the variables
    - `X in 1..5 ∧ Y in 2..6 ∧ X > Y → X in 3..5 ∧ Y in 2..4`
  - Operations: pointwise expansion of regular operations
    - `X in 1..5 ∧ Y in 2..6 ∧ Z #= X + Y → Z in 3..11`
  - Changes to the domains are propagated to other variables
- A possible strategy:
  - Go through the equations one by one
  - Update domains of variables
  - Finish when no more updates

Be a Solver (III)

Step	a	b	c	d	e	f	g
0	0..10	0..10	0..10	0..10	0..10	0..10	0..10
1		0..9			1..10		
2			0..8		2..10		
3						2..10	
4				0..7		3..10	
5					2..6		6..10
6						3..9	
7	0..7						
8		0..5					
9			0..4				
10				0..6			
11	0..4						

Be a Solver (IV)

- Different update strategies can have different performance
- Differences w.r.t. CPM (Critical Path Method):
  - A particular application of finite domains
  - Gives more information than CPM
  - Finding slack for intermediate jobs: simply set $e = 6$, start again —same algorithm, same solver, same representation

  ⇒ Try it!
  - Modeling other relationships without special algorithms:
    - b and c cannot start at the same time: $b \neq c$
    - A task can vary in length devoting different resources; how does this affect the overall project? ($f + x \leq g$)
    - Manpower can be shared between tasks:

$$b + x \leq e \wedge d + y \leq f \wedge x + y = 6$$

Don't Be a Solver

- CLP languages provide built–in solvers
- For a variety of domains
- We will first briefly have a look at
  - Logical variables
  - Backtracking

  using logic programming syntax
- Then we will merge it with constraint solving capabilities
- After this, we will look at Prolog, for practical reasons
- We will finish with a review of the operational model (and pragmatics)
- Everything spiced with examples and questions

A Basic Language

A Basic Constraint Language

- We will use a basic language whose components are:
  - *Variables*, starting with uppercase: $X, Y, Speed$
  - *Constants*, either numbers or names starting with lowercase: $1, 199, cat, a_dog$
    Those constants are the *Domain* of the language
  - *Atoms*, which have the form $p(X_1, \ldots, X_n)$, where:
    - p is the name of the atom, and n its arity (i.e. it admits n arguments) —usually written p/n
    - $X_1, \ldots, X_n$ are either constants or variables
    - Example:

      $hates(dog, cat)$
      $predates(big_fish, small_fish)$
  - *Constraints*: we will use only $= /2$ (syntactic equality)
- We will augment this language later

Clauses

- A *clause* is a construction of the form $p \leftarrow b_1, \ldots, b_n$.
- p is an atom, each of $b_1, \ldots, b_n$ are either *atoms* or *constraints*
- Can be read as: p is true if $b_1, \ldots, b_n$ are all true
- p is termed as the *head*, and $b_1, \ldots, b_n$ as the *body*
- For convenience, $\leftarrow$ is often written as `:-`
- Example:

```
animal(X):- X = dog.
likes(C, F):- C = cat, F = fish.
bigger(M1, M2):- M1 = men, M2 = mice.
```

- Meaning (under a particular interpretation):

  "dog" is an animal *"cat" likes "fish"*

  M1 is bigger than M2 if M1 equals "men" and M2 equals "mice", or *"men" are bigger than "mice"*

Clauses (Cont.)

- Clauses can contain calls to *user predicates*
- Variables are used to pass arguments to them
- Example:

```
eats(X, Y):- bigger(X, Y).
pet(X):- animal(X), sound(X,Y), Y=bark.
```

- Interpretation:

  The big eat the small, or
  If some X is bigger than some Y, then X eats Y

  For X to be a pet, it must be an animal and the sound it produces must be a bark, or
  If X is and animal and X barks, then X is a pet, or
  A pet is an animal which barks

Implicit Equality

- Equality constraints are commonly written in a shorter form
- Ex.: `p(X):- X = something.` can be written as `p(something).`
- I.e., the symbol for the equality constraint does not appear explicitly
- Thus, clauses like

```
bigger(M1, M2):- M1 = men, M2 = mice.
pet(X):- animal(X), sound(X, Y), Y = bark.
```

  are commonly written as

```
bigger(men, mice).
pet(X):- animal(X), sound(X, bark).
```

  with the same meaning as before

Facts

- A *fact* is an expression of the form

$$p.$$

  where p is an atom
- A fact is a clause without body (because the equality constraints are implicitly understood)
- Its meaning is:
  the proposition (relationship) p always holds
- The following are facts:

```
animal(dog).
likes(cat, fish).
bigger(men, mice).
```

- `=/2` can also be defined (again) as `=(X, X).`

Predicates

- A collection of clauses with the same head name
- They represent different possibilities for something to be true
- Or, alternatively, different ways of accomplishing a task
- Example:

```
pet(X):- animal(X), sound(X, bark).
pet(X):- animal(X), sound(X, bubbles).
```

- Meaning:
  A pet is an animal which barks, or an animal which makes bubbles
- Variables in a clause are local to that clause (e.g., the `X`s in the predicate above)

Programs

- A *program* is a set of predicates
- Example:

```
pet(X):- animal(X), sound(X, bark).
pet(X):- animal(X), sound(X, bubbles).

animal(spot).
animal(barry).
animal(hobbes).

sound(spot, bark).
sound(barry, bubbles).
sound(hobbes, roar).
```

⇒ Introduce this program in a (C)LP compiler or interpreter, and play with it (see the next slide)

Making Queries

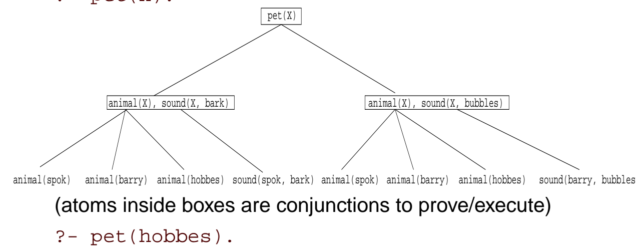
- We interact with a logic program by making *queries*
- A query is a conjunction of atoms
  (which we want to be prove)
- An answer will consist of bindings for the variables in the query which make it true w.r.t. the program
- Several (different) answers are possible
- We will mark a query starting with `?-`
- Example:

```
?- sound(spot, X).
   X = bark
?- animal(X).
   X = spot ;
   X = barry ;
   X = hobbes
```

```
?- sound(A, roar).
   A = hobbes
?- animal(barry).
   yes
?- sound(A, S).
```


Searching (I)

- Automatic search is performed through all paths given by the program
- Is `pet(X)` provable from the program, or
  is there any `X` such that it is a pet?
  `?- pet(X).`



(atoms inside boxes are conjunctions to prove/execute)

`?- pet(hobbes).`

Searching (Cont.)

- Searching allows finding all solutions to a query
- Backtracking is performed:
  - right to left, among goals
  - top to bottom, among clauses

  (but other strategies also possible)

```
?- pet(X), animal(Y).
X = spot, Y = spot ;
⋮
X = barry, Y = hobbes
```

- After the first solution:
  - All possibilities for `animal/1`are tried in the same order as the clauses in the program
  - Then the next clause for `pet/1`is tried
  - And so on, until all paths have been explored

Logical Variables

- Logical variables provide argument passing (to and from predicates)
- And more interesting stuff (we will see later)
- Logical variable assignment is monotonic

```
?- X = a.
X = a
?- X = a, X = b.
no
```

- `X` cannot be at the same time `a` and `b`
- A hint as to why `x := x + 1` $\leftrightarrow x_i := x_{i+1} + 1$ in some imperative languages with constraints (first part of the lectures)

Logical Variables (Cont.)

- The constraint =/2 forces variables to have the same value —i.e., they are *unified*

```
?- X = Y, X = a.
X = a, Y = a.
?- X = Y, pet(X), sound(Y, roars).
no
```

⇒ (why?)

```
?- X = Y, pet(X).
X = spot, Y = spot;
X = barry, Y = barry
```


Logical Variables (Contd.)

- Assume the following program:

```
father_of(juan, pedro).        father_of(juan, maria).
father_of(pedro, miguel).      mother_of(maria, david).
grandfather_of(L,M):- father_of(L,N), father_of(N,M).
grandfather_of(X,Y):- father_of(X,Z), mother_of(Z,Y).
```

⇒ Answer the questions:

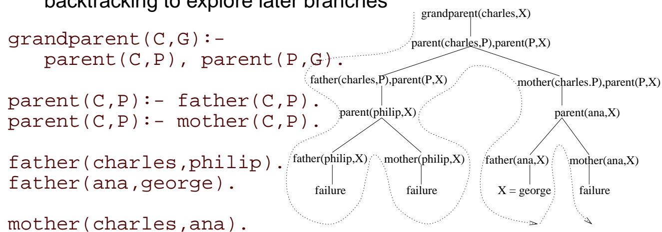
```
?- father_of(juan, pedro).
?- father_of(juan, david).
?- father_of(juan, X).
?- grandfather_of(X, miguel).
?- grandfather_of(X, Y).
?- X = Y, grandfather_of(X, Y).
?- grandfather_of(X, Y), X = Y.
```

Write rules for `grandmother(X, Y)`

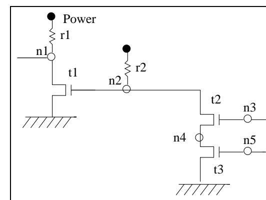
The Execution Mechanism

- Can also be seen as a tree traversal
- Always select the leftmost goal
- Explore the search tree taking the leftmost unexplored branch, backtracking to explore later branches

```
grandparent(C,G):-
   parent(C,P), parent(P,G).

parent(C,P):- father(C,P).
parent(C,P):- mother(C,P).

father(charles,philip).
father(ana,george).

mother(charles,ana).
```



Database Programming: Electrical Circuits




```
resistor(power,n1).
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
inverter(Input,Output):-
   transistor(Input,ground,Output),
   resistor(power,Output).
nand_gate(Input1,Input2,Output):-
   transistor(Input1,X,Output),
   transistor(Input2,ground,X),
   resistor(power,Output).
and_gate(Input1,Input2,Output):-
   nand_gate(Input1,Input2,X),
   inverter(X, Output).
```

```
?- and_gate(In1,In2,Out).
   In1=n3, In2=n5, Out=n1
```

Datalog

- The basic language we have seen so far is a constraint language with:
  - Variables,
  - Constants (the domain of the constraint system),
  - Atoms (user predicates),
  - And a unique constraint $=/2$ which expresses equality between constants or variables
- There are no data structures
- This language, plus numbers, some elementary arithmetic operations, and some other facilities is named *Datalog*, and often used in deductive databases
- Basic operations of relational databases easily expressed!

Datalog and the Relational DB Model

Traditional →	Codd's Relational Model	
File	Relation	Table
Record	Tuple	Row
Field	Attribute	Column

- Example:

Name	Age	Sex
Brown	20	M
Jones	21	F
Smith	36	M

Person

Name	Town	Years
Brown	London	15
Brown	York	5
Jones	Paris	21
Smith	Brussels	15
Smith	Santander	5

Lived–in

- The order of the rows is immaterial
- (Duplicate rows are not allowed)

Datalog and the Relat. DB Model (Contd.)

Relat. Database	→	Logic Programming
Relation Name	→	Predicate symbol
Relation	→	Predicate consisting of ground facts (facts without variables)
Tuple	→	Ground fact
Attribute	→	Argument of predicate

- Examples:

```
person(brown,20,male).
person(jones,21,female).
person(smith,36,male).
```

```
lived_in(brown,london,15).
lived_in(brown,york,5).
lived_in(jones,paris,21).
lived_in(smith,brussels,15).
lived_in(smith,santander,5).
```


Datalog and the Relat. DB Model (Contd.)

- *Union*: r_union_s($X_1,\ldots,X_n$) ←r($X_1,\ldots,X_n$).
  r_union_s($X_1,\ldots,X_n$) ←s($X_1,\ldots,X_n$).
- *Set Difference* (discussion on negation postponed):
  r_diff_s($X_1,\ldots,X_n$) ←r($X_1,\ldots,X_n$),
  not s($X_1,\ldots,X_n$).
  r_diff_s($X_1,\ldots,X_n$) ←s($X_1,\ldots,X_n$),
  not r($X_1,\ldots,X_n$).
- *Cartesian Product*:
  r_X_s($X_1,\ldots,X_m,X_{m+1},\ldots,X_{m+n}$) ←
  r($X_1,\ldots,X_m$), s($X_{m+1},\ldots,X_{m+n}$).
- *Projection*: r13(X_1,X_3) ←r(X_1,X_2,X_3).
- *Selection*:
  r_selected(X_1,X_2,X_3) ←r(X_1,X_2,X_3), $\leq$(X_2,X_3).

Datalog and the Relat. DB Model (Contd.)

- Duplicates an issue: see "setof" later
- Deductive databases use these ideas to develop logic-based databases
  - Often a subset of definite (Prolog) programs used (e.g. "Datalog" – no data structures, no existential variables)
  - Variations of a "bottom-up" execution strategy used (instead of the top-down, goal directed strategy shown here)
  - Formalization use the *immediate consequence* T_p operator to compute a program model, restrict answers to the query

Adding Computation Domains: CLP Programs

What's in a Domain

- Informally, a computation domain for a constraint language comprises:
  - A definition of the elements in the domain
  - Their interpretation,
  - The operations allowed among them, and
  - The interpretation of these operations
- Example:
  - We assumed a finite number of elements in the *precedence net* example
  - A continuous set would not have allowed us to solve the problem as we did

Different Constraint Domains: What for?

- Why different computation domains?
- After all, everything is bytes...
- But:
  - Different problems and elements involved in them have different characteristics
  - Some domains may be better suited for a given problem
  - And people do perceive different properties in the elements around us
- Having different domains allows us to:
  - Abstract the properties of the elements we are working with
  - Choose an appropriate constraint solver
- Thus, we do not have to deal with low–level details
- Example: having operations over complex numbers defined

Constraint Logic Programs

- We will show some examples of constraint logic programs
- We will use different constraint domains to highlight their characteristics
- First we will present the characteristics of the domains and the primitives used
- Then we will model (simple) problems and solve them using CLP

Equations in a Logic Language

- The CLP($\Re$) language and system pioneered linear (dis)equations solving together with a robust logic language
- CLP($\Re$) featured an internal solver for linear equations, including incremental solving thereof
- Equations represented directly as terms (i.e., as written down in paper):

```
?- X = 1 + Y - Z, X > Z, Z = Y + 3.
   Y = Z - 3, X = -2, Z < -2
```

- Many other logic-based languages follow the same path: Prolog II, Prolog III, Prolog IV, SICStus, Ciao Prolog, CHIP, CLP(FD), ...

Linear (Dis)Equations

- A word of caution: syntax details may differ among different implementations
- We will augment the basic language with:
  - Floating point numbers: $4.0, 1e-3$
    (they stand for the corresponding real numbers)
  - Arithmetic operators $(+, *, -, /)$, with which we can construct *arithmetic terms*: $3+4$, $X+3*Y-Z/5$
    These terms stand for arithmetic expressions
  - A predefined constraint $=/2$, which stands for arithmetic equality (can also appear implicitly)
  - Some constraints which represent an order relation: *>=/2, >/2, =</2, </2*
- We will augment the semantics of the language adding constraint solving capabilities

Interacting

- Let us make some queries concerning linear arithmetic

```
?- 4 - Y = 3.
   Y = 1
?- X = 3*Y - X/2, X+Y = Y - 4*X + 7.
   Y = 7/10, X = 7/5.
```

- Sometimes giving a uniquely defined solution is not possible:

```
?- X = Y + 3.
   X = Y + 3
```

- The level of answer definiteness may differ among implementations:

```
?- X >= Y, Y >= X.        ?- X >= Y, Y >= X.
   X = Y                      X - Y >= 0, Y - X >= 0
```

- (Both are *correct*, though)

Linear Problems

- The folowing simple, recursive program defines the natural numbers:

```
nat(0).
nat(N):- N > 0, nat(N-1).
```

(zero is a natural number, and a number greater than zero is also a natural number, provided that its predecessor is a natural number)

```
?- nat(X).              ?- nat(3.4).
   X = 0 ;                 false
   X = 1 ;              ?- nat(-8).
   X = 2 ;                 false
```

Linear Problems

- Even numbers:

```
even(0).
even(N+2):- N+2 > 0, even(N).
```

- The number e: $\frac{e}{4} = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$ (converges very slowly)

```
is_E(N, E4*4):- is_E(N, 1, 1, E4).

is_E(0, Mult, Sign, 0).
is_E(N, Mult, Sign, Sign/Mult + RestE):-
    N > 0,
    is_E(N-1, Mult+1, -Sign, RestE).
```

⇒ Find out wether:
- A number is odd
- A number is a multiple of some other number
- A number N is congruent modulo K to some other number M ($M \equiv N \bmod K$). E.g.: $12 \equiv 7 \bmod 5$, $32 \equiv 2 \bmod 2$, $32 \equiv 2 \bmod 30, \ldots$

Fibonacci

- Fibonacci series:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{n+2} = F_{n+1} + F_n$$

- A straightforward translation:

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

- Find out Fibonacci numbers:

```
?- fib(10, F).
   F = 55.
```

- Find out indexes of Fibonacci numbers:

```
?- fib(N, 8).
   N = 6
```


Fibonacci (Cont.)

- Which are the fixpoints of the Fibonacci series?

```
?- fib(N, N).
   N = 0;
   N = 1;
   N = 5
```

- But this program can break for not-so-large computations:

```
?- fib(100, F).
error: whatever
?-
```

- Solution:
  - Increase available memory
  - Make a better program (too many recursive calls?)

⇒ Develop a simply recursive program

⇒ Find out the 1000th Fibonacci number (last four digits: 8875)

Finite Domains

- Another interesting domain is that of finite domains
- The solver associates to each variable a set of numbers, usually drawn from the naturals
- The (linear) constraint symbols we have seen are extended to work with finite domains
- Operations are pointwise extensions of the corresponding ones in numbers
- We will assume the right semantics and behavior is deduced from the context and variables

Examples of FD Arithmetic

```
?- X = A + B, A in 1..3, B in 3..7.
   B in 3..7, A in 1..3, X in 4..10.
```

- The respective minima and maxima are added

```
?- X = A - B, A in 1..3, B in 3..7.
   B in 3..7, A in 1..3, X in -6..0.
```

- 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 = A - B, A in 1..3, X => 0, B in 3..7.
   B = 3, A = 3, X = 0.
```


Some Useful Primitives

- Being able to access the bounds of a variable is useful
- `fd_min(X, T)`: `T` is the minimum value in the domain of the variable `X` (resp. `fd_max/2`)
- This can be used to minimize (c.f., maximize) a solution

```
?- X = A - B, A in 1..3, B in 3..7, fd_min(X,X).
   B = 7, A = 1, X = -6
```

- Not accurate: propagation-generated domains are *upper bounds*
- We will come to this later

Limits of FD arithmetics

- FD constraints are (partially) solved by means of *propagation*
- A counterpart of algebraic manipulation
- Propagation is a deterministic procedure —does not perform search
- This is **necessarily** not complete:
  - No restriction on what equations can be written
  - Inexistence of general solving methods for, e.g., polinomials of degree 5 (and above)
- Fortunately, the FD domain is *finite* and *discrete*

Enumeration Primitives

- A family of predicates enumerate the values of a variable's domain → causes further simplifications

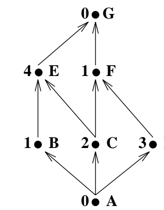
`labeling(Var)`

- instantiates `Var` to all the values in its domain
- Usually available also as `labeling(ListOfVars)`(straightforward to implement)

```
?- Num=1001, Num = A * B, A in 1..Num,
      B in 1..A, labeling([A, B]).
   B = 1, A = 1001, Num = 1001;
   B = 7, A = 143, Num = 1001;
   B = 11, A = 91, Num = 1001;
   B = 13, A = 77, Num = 1001
```

⇒ Try with bigger numbers, and interchanging the order of the enumeration. Is there any difference? Why?

A Project Management Problem (I)



- The job whose dependencies and task lengths are shown at the left should be finished in 10 time units or less
- Constraints:

```
pn1(A, B, C, D, E, F, G):-
    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 + 1.
```

A Project Management Problem (II)

- Query:

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

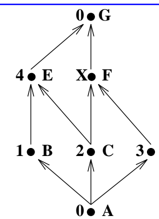
- Note the slack of the variables
- Minimize the total project time:

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

- Variables without slack represent critical tasks

A Project Management Problem (III)

- An alternative setting:



- We can accelerate task **F** at some cost

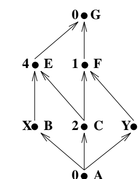
```
pn2(A, B, C, D, E, F, G, X):-
    A >= 0, G =< 10, B >= A, C >= A, D >= A,
    E >= B + 1, E >= C + 2, F >= C + 2,
    F >= D + 3, G >= E + 4, G >= F + X.
```

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

```
?- pn2(A, B, C, D, E, F, G, X), fd_min(G,G).
   X = 3, G = 6, F = 3, E = 2, D = 0,
   C = 0, A = 0, B in 0..1
```


A Project Management Problem (IV)

- We have two independent tasks **B** and **D** whose length is 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:

```
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:

```
?- pn3(A,B,C,D,E,F,G,X,Y), fd_min(G,G).
   Y = 4, X = 2, G = 6, E = 2, C = 0, B = 0,
   A = 0, F in 4..5, D in 0..1
```

- I.e., we must devote more resources to task `X`
- All tasks but `F` and `D` are critical now

Minimization

- In some cases, `fd_min/2` (c.f., `fd_max/2`) is not enough to provide the best solution
- For example, when (incomplete) propagation is unable to find out the tighter bounds

```
?- X in 0..1, Z = X*X, Z \= X, fd_max(Z, W).
   W = 1, X in 0..1, Z in 0..1
```

  - Propagation should not prune more than strictly necessary (risk of missign solutions!)
  - It may prune less than possible → consistency algorithms
- A labeling + branch–and–bound procedure is usually provided:

```
?- pn3(A,B,C,D,E,F,G,X,Y),
   minimize(labeling(G),G).
```


Other Constraints and Operations

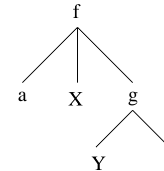
- CLP(FD) systems usually include several more constraint operations / relationship primitives:
  - Disjoint intervals: `A in 1..3 \/ 5..9`
  - Boolean pseudo-operations (based, e.g., on specializing FD to 0..1)
  - Arbitrary operations (suspended until propagation / enumeration takes place)
  - Reification: `C \#<=> B`(`C` constraint, `B` variable):
    - `C` posted if `B = 1`
    - ¬`C` posted if `B = 0`
    - `B = 0` if `C` entailed
    - `B = 1` if `C` disentailed
  - Raises the question of negation and negation in the realm of constraints

Herbrand Terms

- Our last example will be *Herbrand terms*
- Herbrand terms, or *finite trees*, are the non-interpreted functions of first-order logic
- In a logic language, they are very useful to build *data structures*

Herbrand Terms (Cont)

- Herbrand terms are defined as:
  - A variable is a Herbrand term
  - A constant is a Herbrand term
  - A functor f with arity n applied to n terms is a Herbrand term: $f(t_1, t_2, \dots t_n)$
- Herbrand terms are a representation of *trees*:
  - Variables are *open* leaves
  - Constants are fully instantiated leaves
  - Function symbols are nodes
- Example: $f(a, X, g(Y, t))$ represents the tree on the right



Herbrand Terms: Syntactic Equality

- The only constraint allowed between Herbrand terms is `=/2`
- Its meaning is *syntactic equality* (unification)
- Roughly speaking:
  - An equation $f(t_1, \dots, t_n) = f(u_1, \dots, u_n)$ is solved by solving $t_1 = u_1, \dots, t_n = u_n$
  - An equation $v = t$, where v is a variable and t is a term which do not contains v is solved with the binding v/t
  - Equations like $t = t$ can be deleted
  - If none of the above can be applied, the initial terms cannot be unified
- Examples:

```
?- X = f(T, a), T = b.
   T = b, X = f(b,a)
?- f(X, g(X, Y), Y) = f(a, T, b).
   X = a, Y = b, T = g(a, b)
```

Structured Data and Data Abstraction

- Data structures are created using terms:

```
course(comp_logic, mond, 19, 21,
    'Manuel', 'Carro', new, 1302).
```

- When is the Computational Logic course?

```
?- course(comp_logic,Day,Start,End,C,D,E,F).
```

- Structured version:

```
course(comp_logic, When, Lecturer, Location):-
    When = hour(mond,19,21),
    Lecturer = lecturer('Manuel','Hermenegildo'),
    Location = location(new,1302).
```

- When is the Computational Logic course?

```
?- course(comp_logic, When, A, B).
```

Constructing Data Structures

- Functors can be thought of as representing non–interpreted functions (i.e., records)
- The main application of functors is as data structure constructors
- Sample program, step by step
- We have a small database of people who are friends:

```
friends(peter, mark).      ?- friends(anna, X).
friends(anna, marcia).        X = marcia ;
friends(anna, luca).          X = luca

                           ?- friends(X, anna).
                              no
```

- No friends?

Constructing Data Structures (Cont.)

- Being friends is symmetric:

```
are_friends(A, B):- friends(A, B).
are_friends(A, B):- friends(B, A).

?- are_friends(anna, X).   ?- are_friends(X, anna).
   X = marcia ;               X = marcia ;
   X = luca                   X = luca
```

- Some of our friends are married:

```
married(couple(peter, anna)).
married(couple(mark, kathleen)).
married(couple(alvin, marcia)).
```

- Note that married defines *couples* of persons

```
?- married(A).
   A = couple(peter,anna) ;
   A = couple(mark,kathleen) ;
   A = couple(alvin,marcia)
```

Constructing Data Structures (Cont.)

- We may want to know who is spouse of who

```
?- married(couple(peter, S)).
   S = anna
?- married(couple(marcia, S)).
   no
```

- Argument order matters, of course
- We must specify what is a spouse

```
spouse(couple(A, B), A).
spouse(couple(A, B), B).
```

- Then we can forget about the order

```
?- spouse(C, peter),          ?- married(C),
   married(C).                   spouse(C, marcia).
   C = couple(peter,anna)        C = couple(alvin,marcia)
```

Constructing Data Structures (Cont.)

- Some couples go out for dinner if...

```
go_out_for_dinner(Ma, Mb):-
    married(Ma),
    married(Mb),
    spouse(Ma, A),
    spouse(Mb, B),
    are_friends(A, B).

?- go_out_for_dinner(A, B).
   A=couple(peter,anna), B=couple(mark,kathleen) ;
   A=couple(peter,anna), B=couple(alvin,marcia) ;
   A=couple(mark,kathleen), B=couple(peter,anna) ;
   A=couple(alvin,marcia), B=couple(peter,anna)}
```

Structuring Old Problems

```
resistor(r1,power,n1).   transistor(t1,n2,ground,n1).
resistor(r2,power,n2).   transistor(t2,n3,n4,n2).
                         transistor(t3,n5,ground,n4).

inverter(inv(T,R),Input,Output) :-
    transistor(T,Input,ground,Output),
    resistor(R,power,Output).

nand_gate(nand(T1,T2,R),Input1,Input2,Output) :-
    transistor(T1,Input,X,Output),
    transistor(T2,Input2,ground,X),
    resistor(R,power,Output).

and_gate(and(N,I),Input1,Input2,Output) :-
    nand_gate(N,Input1,Input2,X),
    inverter(I,X).

?- and_gate(G,In1,In2,Out).
   G=and(nand(t2,t3,r2), inv(t1,r1)), In1=n3, In2=n5, Out=n1
```

Constructing Recursive Data Structures

- Functions allow constructing recursive data structures
- Simplest recursive data structure: Peano natural numbers
  - `z` is a Peano number (meaning zero, 0)
  - `s(N)` is a Peano number if N is a Peano number (meaning the successor of N, i.e., $N+1$)
- Logic characterization of Peano numbers:

```
natural(z).
natural(s(N)):- natural(N).
```

- Note: very similar to the constraint program

Constructing Recursive Data Structures (Cont.)

```
?- natural(z).
   yes
?- natural(potato).
   no
?- natural(s(s(s(z)))).
   yes
?- natural(X).
   X = z ;
   X = s(z) ;
   X = s(s(z));
   ...
?- natural(s(s(X))).
   X = z ;
   X = s(z) ;
   X = s(s(z));
```


Constructing Recursive Data Structures (Cont.)

- Addition:

```
plus(z, X, X):- natural(X).
plus(s(N), X, s(Y)):- plus(N, X, Y).
```

- Some queries:

```
?- plus(s(s(z)), s(z), R).
   R = s(s(s(z)))
?- plus(s(s(s(z))), T, s(s(s(s(z))))).
   T = s(z)
?- plus(s(s(s(s(z)))), T, s(s(s(z)))).
   no
?- plus(X, Y, s(s(z))).
   X = z, Y = s(s(z)) ;
   X = s(z), Y = s(z) ;
   X = s(s(z)), Y = z}
```


Revisiting Old Friends

- Even numbers, using Peano arithmetic:

```
even(z).
even(s(s(N))):- even(N).
```

⇒ Compare it with the previous constraint version: Where is the `>/2`constraint?

⇒ Try to define: `times(X,Y,Z)`$(Z = X * Y)$, `exp(N,X,Y)`$(Y = X^N)$, `factorial(N,F)` $(F = N!)$, `minimum(N1,N2,Min)`, `ltn(X,Y)`$(X < Y)$

- Define **mod**`(X,Y,Z)` (∃ Q s.t. X = Y*Q + Z and Z < Y):

```
mod(X,Y,Z):- Z < Y, times(Y,Q,W), plus(W,Z,X).
```

- Another possible definition:

```
mod(X,Y,X):- X < Y.
mod(X,Y,Z):- plus(X1,Y,X), mod(X1,Y,Z).
```

- Compare the size of the proof trees

Constructing Recursive Data Structures: Lists

- Lists:
  - A nonempty list is a *head* followed by a tail (another list)
  - Or the *empty* list (nil)
- The functor name usually associated with lists is '.'
- The constant name used to denote the empty list is `[]`
- E.g., `.(a, .(b, .(c, [])))`is the list comprised by the elements `a`, `b`, and `c`
- When some term is a list:

```
is_list([]).
is_list(.(Head, Tail)):- is_list(Tail).
```


Constructing Recursive Data Structures: Lists

- The dot is overloaded → `.(X,Y)`denoted by `[Head|Tail]`

Formal object	Cons pair syntax	Element syntax
.(a,[ ])	[a|[ ]]	[a]
.(a,.(b,[ ]))	[a|[b|[ ]]]	[a,b]
.(a,X)	[a|X]	[a|X]
.(a,.(b,X))	[a|[b|X]]	[a,b|X]

- Note that:
  - `[a,b]` and `[a|X]` unify with `X = [b]`
  - `[a]` and `[a|X]` unify using `X = []`
  - `[a]` and `[a,b|X]` do not unify
  - `[]` and `[X]` do not unify
- Using this notation (note typing!):

```
is_list([]).
is_list([Head|Tail]):- is_list(Tail).
```

Using Recursive Data Structures: Lists

- Definition of membership:

```
member(Element, [Element|List]).
member(Element, [AnElement|RestList]):-
    member(Element, RestList).

?- member(b, [a, b, c]).
   yes
?- member(plof, [a, b, c]).
   no
?- member(X, [a, b, c]).        ?- member(a, X).
   X = a ? ;                        X = [a|_] ? ;
   X = b ? ;                        X = [_, a|_] ? ;
   X = c ?                          X = [_, _, a|_] ? ;
?- member(a, [a, X, c]).            ...
   true ;
   X = a
```


More on Lists

- A useful predicate: `append/3`
- `append(X, Y, Z)`: `Z` is the concatenation of the lists `X` and `Y`

```
append([], X, X).
append([X|Xs], Ys, [X|Zs]):- append(Xs, Ys, Zs).
```

- As usual, it can be used in multiple ways:

```
?- append([1,2,3], [g, h, t], L).
   L = [1,2,3,g,h,t]
?- append(T, [g, h, t], [0, m, g, X, Y]).
   Y = t, X = h, T = [0,m]
?- append(X, Y, [f, p, r]).
   Y = [f,p,r], X = [];
   Y = [p,r], X = [f];
   Y = [r], X = [f,p];
   Y = [], X = [f,p,r]
```

- Different solutions can be enumerated on backtracking

Constraints on Lists

- Some CLP languages (e.g., Prolog IV) include constraints on lists: `size/2`, `index/2`, `o/3`(concatenation)
- `o/3`, as other constraints, does **not** enumerate

```
?- Z = [1,2,3] o [4,5,6].
   Z = [1,2,3,4,5,6].
?- [1,2,3,4,5,6] = X o [4,5,6].
   X = [1,2,3]
?- [1,2,3,4,5,6] = X o Y.
   X o Y = [1,2,3,4,5,6]
```

- But it does constrain:

```
?- [1|Xs] = Xs o [_], size(Xs, 4).
   Xs = [1,1,1,1].
```

⇒ Use `append/3` to make the previous query

- Constraint & generate vs. generate & test

Recursive Programming: Lists (Contd.)

- `reverse(Xs,Ys)`: `Ys` is the list obtained by reversing the elements in the list `Xs`. For each element `X` of `Xs`, put `X` at the end of the rest of the `Xs` list already reversed:

```
reverse([X|Xs],Ys ):-
    reverse(Xs,Zs), append(Zs,[X],Ys).
```

- How can we stop? `reverse([],[]).`
- As defined, `reverse(Xs,Ys)` is very inefficient (why?)
- Another possible definition would be:

```
reverse(Xs,Ys):- reverse(Xs,[],Ys).
reverse([],Ys,Ys).
reverse([X|Xs],Acc,Ys):- reverse(Xs,[X|Acc],Ys).
```

⇒ Find efficiency differences between these two versions

Problems on Lists

- Lists: the most widely used data structures in (C)LP
- Write definitions for the following predicates (use Peano):
  - ⇒ `len(N, L)`: `N` is the length of `L`
  - ⇒ `suffix(S, L)`: `S` is a suffix of `L`
  - ⇒ `prefix(P, L)`: `P` is a prefix of `L`
  - ⇒ `sublist(S, L)`: `S` is a sublist of `L`
  - ⇒ `last(E, L)` `E` is the last element of `L`
  - ⇒ `palindrome(L)`: `L` is a palindrome
  - ⇒ `evenodd(L, E, O)`: `E` is the list of elements in even position in `L` and `O` is the list of the elements in odd position
  - ⇒ `select(E, L1, L2)`: `L2` is `L1` without the element `E`, e.g.,

```
?- select(X, [a,c,n], L).
   L = [c, n], X = a ;
   L = [a, n], X = c ;
   L = [a, c], X = n
```


Constructing Recursive Data Structures: Trees

- A binary tree with a piece of information in each node
- Use, for example, the functor `tree/3`
  - Its first argument is the item of information
  - The second and third ones are the left and right sons (subtrees)
  - Use the constant `void` for empty trees

```
is_tree(void).
is_tree(tree(Info, Left, Right):-
    is_tree(Left), is_tree(Right).
```

- Membership:

```
tree_member(Element, tree(Element, L, R)).
tree_member(Element, tree(E, L, R)):-
    tree_member(Element, L).
tree_member(Element, tree(E, L, R)):-
    tree_member(Element, R).
```

Recursive Data Structures: Trees

- `pre_order(Tree,Order)`: `Order` is a list containing all the elements in `Tree` in preorder

```
pre_order(void,[]).
pre_order(tree(X,Left,Right),Order):-
    pre_order(Left,OrdLft),
    pre_order(Right,OrdRght),
    append([X|OrdLft],OrdRght,Order).
```

⇒ Define:
- `in_order(Tree,Order)`
- `post_order(Tree,Order)`

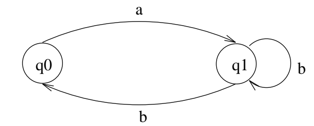
Manipulating Symb. Expressions

- Towers of Hanoi
- Only one disk can be moved at a time
- Larger disks can never be placed on top of a smaller disks
- The predicate is `hanoi(N,A,B,C,Moves)`
- `Moves`: sequence of moves from peg A to peg B using peg C as auxiliary
- Each functor `move(A, B)` represents that the top disk in A goes to B.

```
hanoi(s(0),A,B,C,[move(A, B)]).
hanoi(s(N),A,B,C,Moves):-
    hanoi(N,A,C,B,M1),
    hanoi(N,C,B,A,M2),
    append(M1,[move(A, B)|M2],Moves).
```


Manipulating Symb. Expressions (Contd.)

Recognize the sequence of characters accepted by the following non-deterministic, finite automaton (NDFA) at the right, where `q0` is the initial and final state



- Strings represented as a list of constants

```
initial(q0).        final(q0).
delta(q0,a,q1).     delta(q1,b,q0).     delta(q1,b,q1).

accept(S):- initial(Q), accept(Q,S).
accept(Q,[]):- final(Q).

accept(Q,[X|Xs]):- delta(Q,X,NewQ), accept(NewQ,Xs).
```

CLP in Practice

- Usual CLP systems include Herbrand terms (the primary Prolog constraint system)
- In fact, usually built as extended Prolog systems
- Thus, they inherit all Prolog capabilities
- Plus the added power and flexibility of constraints
- A big advantage: mixing several constraint systems in the same program
- Communication among solvers a problem
- Typing also an issue
- Some allow the user to define user constraints (and even constraint solvers → CHR)
- Herbrand terms + a constraint system: common combination

Linear Equations

- Vector × vector multiplication (dot product):

$$\cdot : \Re^n \times \Re^n \longrightarrow \Re$$
$$(x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = x_1 \cdot y_1 + \cdots + x_n \cdot y_n$$

- Vectors represented as lists of numbers

```
prod([], [], 0).
prod([X|Xs], [Y|Ys], X * Y + Rest):-
    prod(Xs, Ys, Rest).

?- prod([2, 3], [4, 5], K).
   K = 23
?- prod([2, 3], [5, X2], 22).
   X2 = 4
```


Analog RLC circuits

- Analysis and synthesis of analog circuits in steady state
- Each circuit is composed either of a simple component, or a connection of simpler circuits
- We will assume subnetworks connected only in parallel and series ⟶ Ohm laws will suffice
- Relate current (`I`), voltage (`V`) and frequency (`W`) in steady state
- Entry point `circuit(C, V, I, W)`: 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)
- Herbrand terms used to provide data structures

Analog RLC circuits (Cont.)

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

```
c_add(c(Re1,Im1), c(Re2,Im2), R):-
    R = c(Re1+Re2,Im1+Im2).
c_mult(c(Re1,Im1),c(Re2,Im2),c(Re3,Im3)):-
    Re3 = Re1 * Re2 - Im1 * Im2,
    Im3 = Re1 * Im2 + Re2 * Im1.
```

- Circuits in series:

```
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:

```
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 (Cont.)

Basic components modeled as a separate units:

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

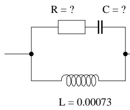
```
circuit(resistor(R), V, I, W):-
    c_mult(I, c(R, 0), V).
```

- Inductor: $V = I * (0 + WLi)$

```
circuit(inductor(L), V, I, W):-
    c_mult(I, c(0, W * L), V).
```

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

```
circuit(capacitor(C), V, I, W):-
    c_mult(I, c(0, -1 / (W * C)), V).
```




```
?- circuit(parallel(inductor(0.073),
                    series(capacitor(C),
                           resistor(R))),
           c(4.5, 0), c(0.65, 0), 2400).
R = 24939720/3608029,
C = 3608029/2365200000.
```

Summarizing

- **In general:**
  - Data structures (Herbrand terms) for free
  - Logical variables may have constraints associated to it
- **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:**
  - Backtracking makes enumeration easy
  - Constraints help keep the search space manageable

Logic Programming: Prolog

- CLP over Herbrand terms is classically referred to as *Logic Programming*
- The programming language Prolog is based on logic programming ideas, plus:
  - Fixed evaluation rules (left–to–right, depth–first)
  - Numerous builtins (input/output, execution control, meta–logic...)
  - Interfacing with external language libraries
  - Efficient implementations
  - . . . and much more
- Prolog can be used advantageously for programming symbolic, complex applications
- Virtually, all modern Prolog implementations provide some sort of constraint capabilities (other than Herbrand terms)

Semantics of CLP Programs

Basic Operations on Constraints

- $\mathcal{D}$ an structure (an interpretation of the symbols of a signature)
- Constraint domains expected to support some basic operations
  1. Consistency (or satisfiability) test: $\mathcal{D} \models \tilde{\exists} c$,
  2. Implication or entailment: $\mathcal{D} \models c_0 \rightarrow c_1$,
  3. Projection of a constraint c_0 onto variables $\tilde{x}$ to obtain a constraint c_1 such that $\mathcal{D} \models c_1 \leftrightarrow \exists_{-\tilde{x}} c_0$,
  4. Detection of uniqueness of variable value: $\mathcal{D} \models c(x, \tilde{z}) \wedge c(y, \tilde{w}) \rightarrow x = y$
- Actually, only the first one is really required
- In actual implementations, some of these operations—in particular the test of consistency—may be incomplete

Basic Operations (II)

- Examples:
  - $x * x < 0$ is inconsistent in $\Re$ (because $\neg\exists x \in \Re : x * x < 0$)
  - $\mathcal{D} \models (x \wedge y = 1) \rightarrow (x \vee y = 1)$ in $\mathcal{BOOL}$
  - In $\mathcal{FT}$, the projection of $x = f(y) \wedge y = f(z)$ on $\{x, z\}$ is $x = f(f(z))$
  - In $\mathcal{WE}$, $\mathcal{D} \models x \cdot a \cdot x = x \cdot a \wedge y \cdot b \cdot y = y \cdot b \rightarrow x = y$
- Prove the last assertion!

Properties of CLP Languages

- $\mathcal{T}$ axiomatizes some of the properties of $\mathcal{D}$
- For a given Σ, let $(\mathcal{D}, \mathcal{L})$ be a constraint domain with signature Σ, and $\mathcal{T}$ a Σ–theory.
- $\mathcal{D}$ and $\mathcal{T}$ correspond on $\mathcal{L}$ if:
  - $\mathcal{D}$ is a model of $\mathcal{T}$, and
  - for every constraint $c \in \mathcal{L}$, $\mathcal{D} \models \tilde{\exists} c$ iff $\mathcal{T} \models \tilde{\exists} c$.
- $\mathcal{T}$ is *satisfaction complete* with respect to $\mathcal{L}$ if for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \tilde{\exists} c$ or $\mathcal{T} \models \neg\tilde{\exists} c$.
- $(\mathcal{D}, \mathcal{L})$ is *solution compact* if

$$\forall c \exists \{c_i\}_{i \in I} : \mathcal{D} \models \forall \tilde{x} \neg c(\tilde{x}) \longleftrightarrow \bigvee_{i \in I} c_i(\tilde{x})$$

  i.e., any negated constraint in $\mathcal{L}$ can be expressed as a (in)finite disjunction of constraints

Solution Compactness

- Important to lift SLDNF results to CLP($\mathcal{X}$)
- We have to deal only with user predicates
- E.g.
  - $x \not\geq y$ in CLP($\Re$) is $x < y$
  - $x \neq y$ in CLP($\Re$) is $x < y \vee y < x$
  - $\Re_{Lin}$ with constraint $x \neq \pi$ is not s.c.
- How can we express $x \neq y$ in CLP($\mathcal{FT}$)?

Logical Semantics (I)

- Two common logical semantics exist.
- The first one interprets a rule

$$p(\tilde{x}) \leftarrow b_1, \ldots, b_n$$

  as the logic formula

$$\forall \tilde{x}, \tilde{y}\ p(\tilde{x}) \vee \neg b_1 \vee \ldots \vee \neg b_n$$

Logical Semantics (II)

Second one: associates a logic formula to each predicate in Π

- If the set of rules of P with p in the head is:

$$\begin{array}{rcl} p(\tilde{x}) & \leftarrow & B_1 \\ & & \vdots \\ p(\tilde{x}) & \leftarrow & B_n \end{array}$$

  then the formula associated with p is:

$$\begin{array}{rcl} \forall \tilde{x}\ p(\tilde{x}) & \leftrightarrow & \exists \tilde{y}_1 B_1 \\ & \vee & \exists \tilde{y}_2 B_2 \\ & & \vdots \\ & \vee & \exists \tilde{y}_n B_n \end{array}$$

- When p does not occur in the head of a rule of P: $\forall \tilde{x} \neg p(\tilde{x})$
- *Clark completion* of P (denoted by P^*)
- These two semantics differ on the treatment of the negation

Logical Semantics (III)

- *Valuation*: a mapping from variables to D
- Also maps terms to D and formulas to closed $\mathcal{L}^*$–formulas.
- $\mathcal{D}$–interpretation of a formula: same domain as $\mathcal{D}$ and same interpretation for the symbols in Σ as $\mathcal{D}$
- Represented as a subset of $B_{\mathcal{D}}$ where

$$B_{\mathcal{D}} = \{p(\tilde{d}) \mid p \in \Pi, \tilde{d} \in D^k\}$$

- $\mathcal{D}$–model of a closed formula: $\mathcal{D}$–interpretation which is a model of the formula.
- The usual logical semantics is based on the $\mathcal{D}$–models of P and the models of P^*, $\mathcal{T}$.
- The least $\mathcal{D}$–model of a formula Q is denoted by $lm(Q, \mathcal{D})$.
- A *solution* to a query G is a valuation v such that $v(G) \subseteq lm(P, \mathcal{D})$.

Fixpoint Semantics

- Based on one-step consequence operator $T_P^{\mathcal{D}}$ (also called "immediate consequence operator").
- Take as semantics $lfp(T_P^{\mathcal{D}})$, where:

$$T_P^{\mathcal{D}}(I) = \{p(\tilde{d}) \quad | \quad p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n \in P, a_i \in I, \\ \mathcal{D} \models v(c), v(\tilde{x}) = \tilde{d}, v(b_i) = a_i\}$$

- Theorems:
  1. $T_P^{\mathcal{D}} \uparrow \omega = lfp(T_P^{\mathcal{D}})$
  2. $lm(P, \mathcal{D}) = lfp(T_P^{\mathcal{D}})$

Top–Down Operational Semantics (I)

- General framework, formalized as a transition system
- State: a 3–tuple $\langle A, C, S\rangle$, or $fail$, where
  - A is a multiset of atoms and constraints,
  - $C \bigcup S$ multiset of constraints,
  - C, active constraints (awake)
  - S, passive constraints (asleep)
- *Computation* and *selection* rules depend on A
- Transition system: parameterized by a predicate $consistent$ and a function $infer$:
  - $consistent(C)$ checks the consistency of a constraint store
  - Usually "$consistent(C)$ iff $\mathcal{D} \models \tilde{\exists} c$", but sometimes "if $\mathcal{D} \models \tilde{\exists} c$ then $consistent(C)$"
  - $infer(C, S)$ computes a new set of active and passive constraints

Top–Down Operational Semantics (II)

- Transition r: rewriting using user predicates
  $\langle A \bigcup a, C, S\rangle \rightarrow_r \langle A \bigcup B, C, S \bigcup (a = h)\rangle$
  if $h \leftarrow B \in P$, and a and h have the same predicate symbol, or
  $\langle A \bigcup a, C, S\rangle \rightarrow_r fail$
  if there is no rule $h \leftarrow B$ of P such that a and h have the same predicate symbol ($a = h$ is a set of argument–wise equations) if a is a predicate symbol selected by the computation rule
- Transition c: selects constraints $\langle A \bigcup c, C, S\rangle \rightarrow_c \langle A, C, S \bigcup c\rangle$
  if c is a constraint selected by the computation rule
- Transition i: infers new constraints
  $\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$ if $(C', S') = infer(C, S)$
  - In particular, may turn passive constraints into active ones
- Transition s: checks satisfiability

$$\langle A, C, S\rangle \rightarrow_s \begin{cases} \langle A, C, S\rangle & \text{if } consistent(C) \\ fail & \text{if } \neg consistent(C) \end{cases}$$

Top–Down Operational Semantics (III)

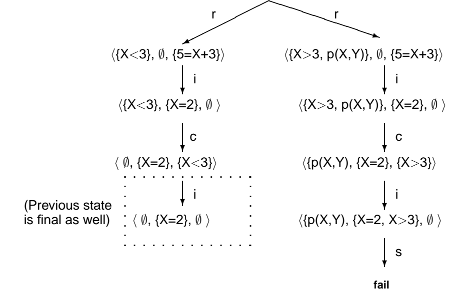
- Initial state: $\langle G, \emptyset, \emptyset\rangle$
- Derivation: $\langle A_1, C_1, S_1\rangle \rightarrow \ldots \rightarrow \langle A_i, C_i, S_i\rangle \rightarrow \ldots$
- Final state: $E \rightarrow E$
- *Successful derivation*: final state $\langle \emptyset, C, S\rangle$
- A derivation *flounders* if finite and the final state is $\langle A, C, S\rangle$ with $A \neq \emptyset$
- A derivation is *failed* if it is finite and the final state is fail
- Answer: $\exists_{-\tilde{x}} C \wedge S$, where $\tilde{x}$ are the variables in the initial goal
- A derivation is *fair* if it is failed or, for every i and every $a \in A_i$, a is rewritten in a later transition
- A computation rule is fair if it gives rise only to fair derivations

Top–Down Operational Semantics (IV)

- *Computation tree* for goal G and program P:
  - Nodes labeled with states
  - Edges labeled with $\rightarrow_r$, $\rightarrow_c$, $\rightarrow_i$ or $\rightarrow_s$
  - Root labeled by $\langle G, \emptyset, \emptyset\rangle$
  - All sons of a given node have the same label
  - Only one son with transitions $\rightarrow_c$, $\rightarrow_i$ or $\rightarrow_s$
  - A son per program clause with transition $\rightarrow_r$

Computation Tree: Example

- Consider the program
  $p(X + 3, X) \leftarrow X < 3.$
  $p(X + 3, X) \leftarrow X > 3, p(X, Y).$
  and the goal $\leftarrow p(5, X)$
- A possible computation tree is:



Types of CLP($\mathcal{X}$) Systems

- *Quick–checking* CLP($\mathcal{X}$) system: its operational semantics can be described by $\rightarrow_{ris} \equiv \rightarrow_r \rightarrow_i \rightarrow_s$ and $\rightarrow_{cis} \equiv \rightarrow_c \rightarrow_i \rightarrow_s$
- I.e., always selects either an atom or a constraint, infers and checks consistency
- *Progressive* CLP system: for all $\langle A, C, S \rangle$ with $A \neq \emptyset$, every derivation from that state either fails or contains a $\rightarrow_r$ or $\rightarrow_c$ transition
- *Ideal* CLP system:
  - Quick-checking
  - Progressive
  - $infer(C, S) = (C \cup S, \emptyset)$
  - $consistent(C)$ holds iff $\mathcal{D} \models \tilde{\exists} c$

Soundness and Completeness Results

- Success set: queries + constraints with successful derivation $SS(P) = \{p(\tilde{x}) \leftarrow c \mid \langle p(\tilde{x}), \emptyset, \emptyset \rangle \rightarrow^* \langle \emptyset, c', c'' \rangle, \mathcal{D} \models c \leftrightarrow \exists_{-\tilde{x}} c' \wedge c''\}$
- Program P in the language determined by 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$ and executing on an **ideal** CLP system. Then:
  1. $[SS(P)]_{\mathcal{D}} = lm(P, D)$, where
     $$[SS(P)]_{\mathcal{D}} = \{v(a) \mid (a \leftarrow c) \in SS(P), \mathcal{D} \models v(c)\}$$
  2. $SS(P) = lfp(S_P^{\mathcal{D}})$
  3. (Soundness) if the goal G has a successful derivation with answer constraint c, then $P, \mathcal{T} \models c \rightarrow G$
  4. (Completeness) if $P, \mathcal{T} \models c \rightarrow G$ then there are derivations for the goal G with answer constraints $c_1, \ldots, c_n$ such that $\mathcal{T} \models c \rightarrow \bigvee_{i=1}^{n} c_i$
  5. Assume $\mathcal{T}$ is satisfaction complete w.r.t. $\mathcal{L}$. Then the goal G is finitely failed for P iff $P^*, \mathcal{T} \models \neg G$.

Negation in CLP($\mathcal{X}$)

- Most LP results can be lifted to CLP($\mathcal{X}$)
- In particular, negation as failure (à la SLDNF) is still valid using:
  - Satisfiability instead of unification
  - Variable elimination instead of groundness
- Added bonus: if the system is *solution compact*, then negated constraints can be expressed in terms of primitive constraints
- Less chances of a floundered / incorrect computation

Complex Constraints

- Some complex constraints allow expressing simpler constraints
- May be operationally treated as passive constraints
- E.g.: cardinality operator $\#(L, [c_1, \ldots, c_n], U)$: the number of true constraints lies between L and U (can be variables)
  - $L = U = n \rightarrow$ all constraints must hold
  - $L = U = 1 \rightarrow$ one and only one constraint must be true
  - Constraining $U = 0$, we force the conjunction of the negations to be true
  - $L > 0 \rightarrow$ the disjunction of the constraints is specified
- Disjunctive constructive constraint: $c_1 \vee c_2$
  - If properly handled, avoids search and backtracking
  - E.g.: $nz(X) \leftarrow X > 0.$ $nz(X) \leftarrow X < 0.$ $\quad nz(X) \leftarrow X < 0 \vee X > 0.$

Implementation Issues: Satisfiability

- Algorithms must be *incremental* in order to be practical
- Incrementality refers to the performance of the algorithm
- It is important that algorithms to decide satisfiability have a good average case behavior
- Common technique: use a *solved form* representation for satisfiable constraints
- Not possible in every domain
- E.g. in $\mathcal{FT}$ constraints are represented in the form $x_1 = t_1(\tilde{y}), \ldots, x_n = t_n(\tilde{y})$, where
  - each $t_i(\tilde{y})$ denotes a term structure containing variables from $\tilde{y}$
  - no variable x_i appears in $\tilde{y}$

  (i.e. idempotent substitutions, guaranteed by the unification algorithm)