

CSCI 561

Foundation for Artificial Intelligence

15. Logic Reasoning Systems

Professor Wei-Min Shen
University of Southern California

Logical Reasoning Systems

(→) Intro

1. 4 types

- **Theorem provers and logic programming languages** – Provers: use resolution to prove sentences in full FOL. Languages: use backward chaining on restricted set of FOL constructs.
- **Production systems** – based on implications, with consequents interpreted as action (e.g., insertion & deletion in KB). Based on forward chaining + conflict resolution if several possible actions.
- **Frame systems and semantic networks** – objects as nodes in a graph, nodes organized as taxonomy, links represent binary relations.
- **Description logic systems** – evolved from semantic nets. Reason with object classes & relations among them.

2.

Basic Tasks

- Add a new fact to KB – TELL
- Given KB and new facts, derive facts implied by conjunction of KB and the new facts. In forward chaining: part of TELL
- Decide if query entailed by KB – ASK
- Decide if query explicitly stored in KB – restricted ASK, no reasoning, just search content of KB
- Remove sentence from KB: distinguish between correcting false sentence, forgetting useless sentence, or updating KB re. change in the world.

3

Indexing, retrieval & unification

↓

- ① **Implementing sentences & terms:** define syntax and map sentences onto machine representation.

Compound: has operator & arguments.

e.g., c = P(x) \wedge Q(x)

Compound sentence

Op operator: \wedge

Op[c] = \wedge ; Args[c] = [P(x), Q(x)]

C_As arguments: P(x), Q(x)

- ② **FETCH:** find sentences in KB that have same structure as query.

ASK makes multiple calls to FETCH.

takes O(n) time on n-element KB

- ③ **STORE:** add each conjunct of sentence to KB. Used by TELL.

e.g., implement KB as list of conjuncts

TELL(KB, A \wedge \neg B) TELL(KB, \neg C \wedge D)

then KB contains: [A, \neg B, \neg C, D]

Faster Solutions = ? do hash

takes O(n) time on n-element KB (if check for duplicates)

STORE(s) stores a sentence s into the knowledge base

FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base

(2) KB in store is like table

Table-Based Indexing

- What are you indexing on? Predicates (relations/functions). Example:

Key	Positive	Negative	Conclusion	Premise
Mother	Mother(ann,sam) Mother(grace,joe)	-Mother(ann,al)	xxxx	xxxx
dog	dog(rover) dog(fido)	-dog(alice)	xxxx	xxxx

- Use hash table to avoid looping over entire KB for each TELL or FETCH

e.g., if only allowed literals are single letters, use a 26-element array to store their values.

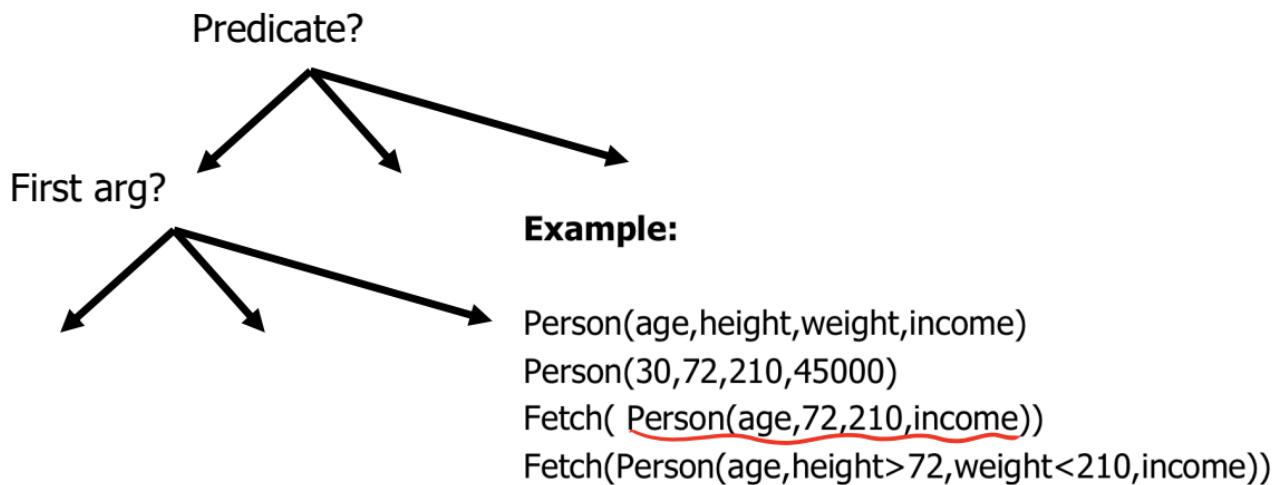
- More generally:

- convert to Horn form
- index table by predicate symbol
- for each symbol, store:
 - list of positive literals
 - list of negative literals
 - list of sentences in which predicate is in conclusion
 - list of sentences in which predicate is in premise

(2)

Tree-Based Indexing

- Hash table **impractical** if many clauses for a given predicate symbol
- **Tree-based indexing** (or more generally combined indexing):
compute indexing key from predicate and argument symbols



(3)

Unification Algorithm

- Using clever indexing, can reduce number of calls to unification
- Still, unification is called very often (at basis of modus ponens) => need efficient implementation.
- See AIMA p. 303 for example of algorithm with $O(n^2)$ complexity (n being size of expressions being unified).

e.g:

Understands(mary,x) implies Loves(mary,x)

Understands(mary,pete) allows the system to
substitute pete for x, and make the implication that

IF Understands(mary,pete) THEN Loves(mary,pete)

Logic Programming

Remember: knowledge engineering vs. programming...

Sound bite: computation as inference on logical KBs

Logic programming

1. Identify problem
2. Assemble information
3. Tea break
4. Encode information in KB
5. Encode problem instance as facts
6. Ask queries
7. Find false facts

Ordinary programming

- Identify problem
- Assemble information
- Figure out solution
- Program solution
- Encode problem instance as data
- Apply program to data
- Debug procedural errors

Should be easier to debug *Capital(NewYork, US)* than $x := x + 2$!

Logic Programming Systems

1.

e.g., **Prolog**: (one language: allows to write sentence in horn form in FOL)

- Program = sequence of sentences (implicitly conjoined)
- All variables implicitly universally quantified
- Variables in different sentences considered distinct
- Horn clause sentences only (= atomic sentences or sentences with no negated antecedent and atomic consequent)
- Terms = constant symbols, variables or functional terms
- Queries = conjunctions, disjunctions, variables, functional terms
- Instead of negated antecedents, use negation as failure operator: goal NOT P considered proved if system fails to prove P
- Syntactically distinct objects refer to distinct objects
- Many built-in predicates (arithmetic, I/O, etc)

Prolog Systems

Basis: backward chaining with Horn clauses + bells & whistles

Widely used in Europe, Japan (basis of 5th Generation project)

Compilation techniques \Rightarrow 10 million LIPS

Program = set of clauses = head :- literal₁, ... literal_n.

Efficient unification by open coding

Efficient retrieval of matching clauses by direct linking

Depth-first, left-to-right backward chaining

Built-in predicates for arithmetic etc., e.g., X is Y*Z+3

Closed-world assumption ("negation as failure")

e.g., `not PhD(X)` succeeds if `PhD(X)` fails

- A PROLOG program is a set of **facts** and **rules**.
- A simple program with just facts :

↓

```
parent(alice, jim) .  
parent(jim, tim) .  
parent(jim, dave) .  
parent(jim, sharon) .  
parent(tim, james) .  
parent(tim, thomas) .
```

- c.f. a table in a relational database.
- Each line is a fact (a.k.a. a tuple or a row).
- Each line states that some person X is a parent of some (other) person Y.
- In GNU PROLOG the program is kept in an ASCII file.

A PROLOG Query

- Now we can ask PROLOG questions :

```
| ?- parent(alice, jim).  
yes      is parent(.,.) true?
```

```
| ?- parent(jim, herbert).  
no  
| ?-
```

- Not very exciting. But what about this :

alice is who's parent?

```
| ?- parent(alice, Who).  
Who = jim  
yes  
| ?-
```

- Who is called a **logical variable**.

- PROLOG will set a logical variable to any value which makes the query succeed.

- Sometimes there is more than one correct answer to a query.
- PROLOG gives the answers one at a time. To get the next answer type ;.

```
| ?- parent(jim, Who).  
Who = tim ? ;  
Who = dave ? ;  
Who = sharon ? ;  
yes  
| ?-
```

NB : The ;
do not
actually
appear on
the screen.

- After finding that jim was a parent of sharon GNU PROLOG detects that there are no more alternatives for parent and ends the search.

2.

Basic syntax of facts, rules and queries

```
<fact> ::= <term> .  
<rule> ::= <term> :- <term> .  
<query> ::= <term> .  
<term> ::= <number> | <atom> | <variable> | <atom>  
(<terms>)  
<terms> ::= <term> | <term>, <terms>
```

Nice very concise intro to prolog:

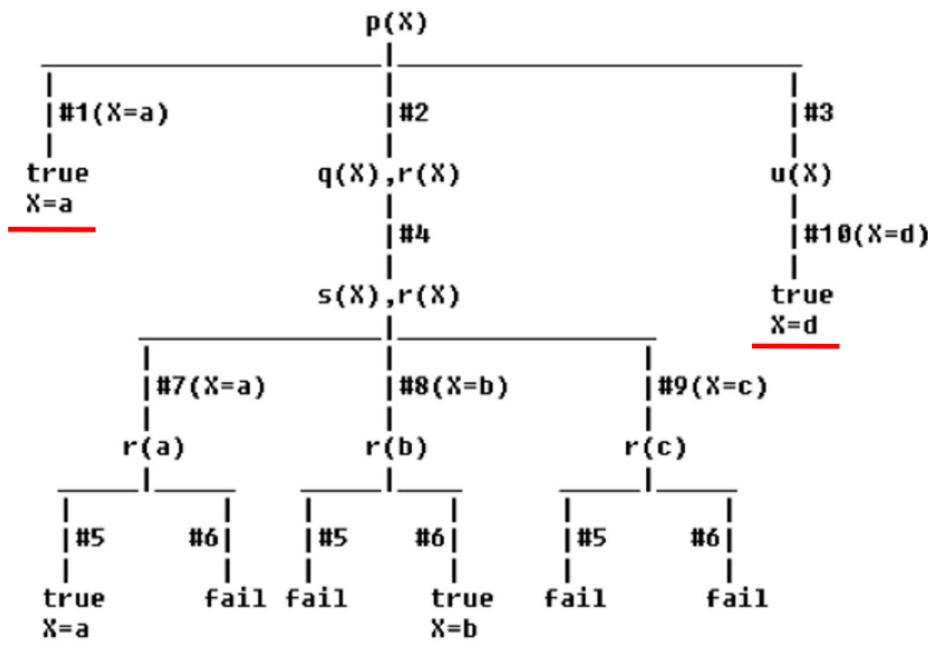
<https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Prolog.html>

Basic syntax of facts, rules and queries

eg :

```
/* program P      q(X) \& r(X) → p(X) clause #      */
p(a).          /* #1 */
p(X) :- q(X), r(X). /* #2 */
p(X) :- u(X).  u(X) → p(X) /* #3 */
q(X) :- s(X).  s(X) → q(X) /* #4 */
r(a).          /* #5 */
r(b).          /* #6 */
s(a).          /* #7 */
s(b).          /* #8 */
s(c).          /* #9 */
u(d).          /* #10 */
```

Query for $p(X)$ (i.e., for which values of X is $p(X)$ true):



3. Prolog Example

Depth-first search from a start state X:

two rules define
same thing.
then can apply on itself,
called recursive definition

```
dfs(X) :- goal(X).  
dfs(X) :- successor(X,S), dfs(S).
```

conjunction

No need to loop over S: successor succeeds for each

4. Appending two lists to produce a third:

```
append([], Y, Y). [ ] .append(Y) = Y  
append([X|L], Y, [X|Z]) :- append(L, Y, Z). [X|L].append(Y) = [X|Z]
```

“cons”

query: append(A,B,[1,2]) ?

answers: A=[] B=[1,2]

A=[1] B=[2]

A=[1,2] B=[]

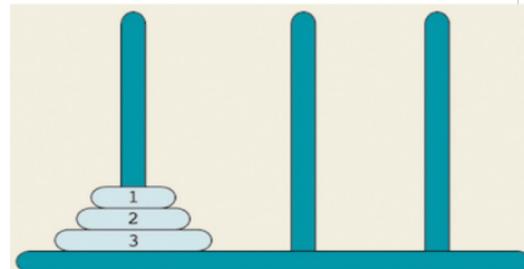
Append

- **append([], L, L)**
- **append([H| L1], L2, [H| L3]) :- append(L1, L2, L3)**
- Example join [a, b, c] with [d, e].
 - [a, b, c] has the recursive structure [a] [b, c].
 - Then the rule says:
 - IF [b, c] appends with [d, e] to form [b, c, d, e],
THEN [a][[b, c]] appends with [d, e] to form [a][[b, c, d, e]]
 - i.e. [a, b, c] [a, b, c, d, e]

eg:

Tower of Hanoi in Prolog

```
% move(N,X,Y,Z) - move N disks from peg X to peg Y, with peg Z being the
% auxilliary peg
%
% Strategy:
% Base Case: One disc - To transfer a stack consisting of 1 disc from
% peg X to peg Y, simply move that disc from X to Y
% Recursive Case: To transfer n discs from X to Y, do the following:
%   Transfer the first n-1 discs to some other peg X
%   Move the last disc on X to Y
%   Transfer the n-1 discs from X to peg Y
%           Anonymous variable _
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl. ← Write a newline
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```



Tower of Hanoi in Prolog

```
% move(N,X,Y,Z) - move N disks from peg X to peg Y, with peg Z being the
%                 auxilliary peg
%
% Strategy:
% Base Case: One disc - To transfer a stack consisting of 1 disc from
%             peg X to peg Y, simply move that disc from X to Y
% Recursive Case: To transfer n discs from X to Y, do the following:
%                 Transfer the first n-1 discs to some other peg X
%                 Move the last disc on X to Y
%                 Transfer the n-1 discs from X to peg Y

move(1,X,Y,_):-  
    write('Move top disk from '),           Here is what happens when Prolog solves the case N=3.  
    write(X),  
    write(' to '),  
    write(Y),  
    nl.  
move(N,X,Y,Z):-  
    N>1,  
    M is N-1,  
    move(M,X,Z,Y),  
    move(1,X,Y,_),  
    move(M,Z,Y,X).
```

?- move(3, left, right, center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

yes

Sudoku in Prolog

The key to solving Sudoku puzzles with Prolog is to use the `clpfd` (constraint logic programming over finite domains) library to restrict the search space to numbers 1-9. Then, it's just a matter of describing what a solution looks like.

Availability: :- use module(library(library)).. (can be autoloaded)

maplist(*:Goal*, *?List*)

8

True if *Goal* can successfully be applied on all elements of *List*. Arguments are reordered to gain performance as well as to make the predicate deterministic under normal circumstances.

Availability: :- use_module(library(clpf)).

+Vars ins +Domain

The variables in the list `Vars` are elements of `Domain`. See [in/2](#) for the syntax of `Domain`.

Sudoku in Prolog

The key to solving Sudoku puzzles with Prolog is to use the `clpfd` (constraint logic programming over finite domains) library to restrict the search space to numbers 1-9. Then, it's just a matter of describing what a solution looks like.

```
%% need the module "clpfd" (constraint logic programming over finite domains)
%% so that we can specify the range of numbers to search
?- use_module(library(clpfd)).

sudoku(Rows) :-
    length(Rows, 9), % ensure there are 9 rows
    maplist(length_(9), Rows), % ensure each row has 9 elements (see below for length_)
    append(Rows, Vs), % combined all rows into the variable Vs
    Vs ins 1..9, % ensure that the elements of Vs should be numbers 1-9
    maplist(all_distinct, Rows), % ensure each row is distinct
    transpose(Rows, Columns), % flip the matrix
    maplist(all_distinct, Columns), % ensure each column is distinct
    Rows = [A,B,C,D,E,F,G,H,I], % create variables A-I for each row
    blocks(A, B, C), % make sure all values in these three rows (three 3x3 blocks) are distinct
    blocks(D, E, F), % ... and these rows/blocks
    blocks(G, H, I). % ... and these rows/blocks
```

Sudoku in Prolog

```
length_(L, Ls) :- length(Ls, L). % version of length that's easier to use with maplist (above)

% this predicate ensures a "block" (3x3 grid) contains only distinct values
blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
    all_distinct([A,B,C,D,E,F,G,H,I]),
    blocks(Bs1, Bs2, Bs3).

problem([[_,_,_,_,_,_,_,_,_],
         [_,_,_,_,_,3,_,8,5],
         [_,_,1,_,2,_,_,_,_],
         [_,_,_,5,_,7,_,_,_],
         [_,_,4,_,_,_,1,_,_],
         [_,9,_,_,_,_,_,_,_],
         [5,_,_,_,_,_,_,7,3],
         [_,_,2,_,1,_,_,_,_],
         [_,_,_,_,4,_,_,_,9]]). solve_problems :-
    problem(Rows),
    statistics(runtime, _), % builtin function, establishes "runtime" variable
    sudoku(Rows), % solve the puzzle
    maplist(writeln, Rows), % show the solution (writeln, i.e., println each row)
    statistics(runtime, [_,T]), % use "runtime" variable to compute total time
    write('CPU time = '), write(T), write(' msec'), nl, nl, false. % write total time
```

Sudoku in Prolog

```
itti@iLab0:~/prolog sudoku.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.
```

```
For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
```

```
?- solve_problems.
[9,8,7,6,5,4,3,2,1]
[2,4,6,1,7,3,9,8,5]
[3,5,1,9,2,8,7,4,6]
[1,2,8,5,3,7,6,9,4]
[6,3,4,8,9,2,1,5,7]
[7,9,5,4,6,1,8,3,2]
[5,1,9,2,8,6,4,7,3]
[4,7,2,3,1,9,5,6,8]
[8,6,3,7,4,5,2,1,9]
CPU time = 44 msec
```

```
false.
```

```
?- □
```

Note: returned false here only
so that the program would
then try to solve any other declared
Instance of problem().

5. Expanding Prolog complete for all *horn clause*

(1) Parallelization:

- { OR-parallelism: goal may unify with many different literals and implications in KB
- AND-parallelism: solve each conjunct in body of an implication in parallel

(2) Compilation: generate built-in theorem prover for different predicates in KB

(3) Optimization: for example through re-ordering

e.g., "what is the income of the spouse of the president?"

$\text{Income}(s, i) \wedge \text{Married}(s, p) \wedge \text{Occupation}(p, \text{President})$

faster if re-ordered as:

$\text{Occupation}(p, \text{President}) \wedge \text{Married}(s, p) \wedge \text{Income}(s, i)$

Reduce search space

(2).

Theorem Provers

- Differ from logic programming languages in that:
 - accept full FOL
 - results independent of form in which KB entered

2. OTTER *System implement theorem prover using Resolution*

- Organized Techniques for Theorem Proving and Effective Research (McCune, 1992)
- **Set of Support (SOS):** set of clauses defining facts about a given problem
- Each resolution step: resolves member of sos against other axiom
- Usable axioms (outside sos): provide background knowledge about domain
- Rewrites (or demodulators): define canonical forms into which terms can be simplified.
E.g., $x+0=x$
- **Control strategy:** defined by set of parameters and clauses. E.g., heuristic function to control search, filtering function to eliminate uninteresting subgoals.
- **Operation:** resolve elements of SOS against usable axioms
- **Use best-first search:** heuristic function measures "weight" of each clause (lighter weight preferred; thus in general weight correlated with size/difficulty)
 - At each step: move lightest clause in SOS to usable list, and add to usable list consequences of resolving that clause against usable list
- **Halt:** when refutation found or SOS is empty

OTTER

Netscape: Otter: An Automated Deduction System

File Edit View Go Communicator Help

Bookmarks Location: <http://www-unix.mcs.anl.gov/AR/otter/> What's Related

Google Gscout Library WoS PubMed INSPEC JTrack ResearchIndex WebAdmin CVSweb HC Params

Otter: An Automated Deduction System

Updated August 13, 2001.

Contents

1. [Description](#)
2. [Computational Environment](#)
3. [Availability](#) Version 3.2 
4. [Documentation](#)
5. [Example Inputs](#) 
6. [Recent Accomplishments](#)
7. [Performance on the TPTP Problems](#)
8. [Bugs and Fixes](#)
9. [Otter-users Mailing List](#)



Related Pages

- Try Otter right now with [Son of BirdBrain](#)
- [A sample Otter proof](#)
- [New Results](#) obtained with Otter and related programs
- [MACE](#), a program that searches for small models
- [EQP](#), a prover for equational logic with associative unification
- [Automated Reasoning at Argonne](#)

External Work

- [Johan Beilinante's Set Theory Work with Otter](#)
- [Some other theorem provers](#)
- [Otter mode for Emacs](#) (from Holger Schauer)
- [GOAL](#) by Guoxiang Huang and Dale Myers
- [A student project on Otter by Jackson Pauls](#)

Description

Our current automated deduction system Otter is designed to prove theorems stated in first-order logic with equality. Otter's inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing

Example: Robbins Algebras Are Boolean

- The Robbins problem---are all Robbins algebras Boolean?---has been solved: Every Robbins algebra is Boolean. This theorem was proved automatically by EQP, a theorem proving program developed at Argonne National Laboratory

<https://www.mcs.anl.gov/research/projects/AR/eqp/>

Example: Robbins Algebras Are Boolean

Historical Background

$ \quad \quad y \quad \quad -x \quad \quad -x \quad \quad $ $ \quad \quad y \quad \quad -x \quad \quad -x \quad \quad $ $ -(y-x)- $ $-(y-x)- $	$0 -x- \quad \quad y - $ $-x-$	$// \rightarrow (x+y)$ $// x + \neg(y)$ $// \neg(x + \neg(y))$
--	--------------------------------	--

- In 1933, E. V. Huntington presented the following basis for Boolean algebra:

$$x + y = y + x.$$

〔commutativity〕

// + is v

$$(x + y) + z = x + (y + z).$$

Associativity

$$n(n(x) + y) + n(n(x) + n(y)) = x.$$

[Huntington equation] // n is -

- Shortly thereafter, Herbert Robbins conjectured that the Huntington equation can be replaced with a simpler one:

$$n(n(x + y) + n(x + n(y))) = x.$$

[Robbins equation]

- Robbins and Huntington could not find a proof, and the problem was later studied by Tarski and his students

Searching ...

Success, in 1.28 seconds!

$\sim\alpha$

----- PROOF -----

1	$n(n(A) + B) + n(n(A) + n(B)) \neq A.$	
2	$x=x.$	
3	$x+y=y+x.$	
5, 4	$(x+y)+z=x+(y+z).$	
6	$n(n(x+y) + n(x+n(y))) = x.$	
8	$x+x=x.$	
10	$n(n(A) + n(B)) + n(n(A) + B) \neq A.$	[para_from, 3, 1]
13	$x+ (x+y) = x+y.$	[para_into, 4, 8, flip.1]
15	$x+ (y+z) = y+ (x+z).$	[para_into, 4, 3, demod, 5]
23, 22	$x+ (y+x) = x+y.$	[para_into, 13, 3]
26	$n(n(x) + n(x+n(x))) = x.$	[para_into, 6, 8]
36	$n(n(n(x) + x) + n(n(x))) = n(x).$	[para_into, 6, 8]
42	$n(n(x+n(y)) + n(x+y)) = x.$	[para_into, 6, 3]
52	$x+ (y+z) = x+ (z+y).$	[para_into, 15, 3, demod, 5]
81, 80	$n(n(x+n(x)) + n(x)) = x.$	[para_into, 26, 3]
82	$n(n(n(x) + x) + x) = n(x).$	[para_from, 26, 6, demod, 23]
125	$n(n(n(x+n(x)) + (n(x) + x)) + x) = n(x+n(x)) + n(x).$	[para_into, 80, 80, demod, 5, 81]
139	$n(n(n(x+n(x)) + x) + x) = n(x+n(x)).$	[para_from, 80, 6]
166, 165	$n(n(x+n(x)) + x) = n(x).$	[para_into, 82, 3]
180, 179	$n(n(x) + x) = n(x+n(x)).$	[back_demod, 139, demod, 166]
195	$n(n(x+n(x)) + n(n(x))) = n(x).$	[back_demod, 36, demod, 180]
197	$n(n(x+ (n(x) + n(x+n(x)))) + (n(x+n(x)) + x)) = n(x).$	[para_into, 165, 165, demod, 5, 180, 5, 166]
206, 205	$n(n(x+ (n(x) + n(x+n(x)))) + n(x)) = n(x+n(x)) + x.$	[para_from, 165, 80, demod, 166, 5, 180, 5]
223, 222	$n(n(x+y) + (y+x)) = n(x+ (y+n(x+y))).$	[para_into, 179, 52, demod, 5]
231, 230	$n(n(x+ (n(x) + n(x+n(x)))) + x) = n(x+n(x)) + n(x).$	[back_demod, 125, demod, 223]
564, 563	$n(x+n(x)) + x = x.$	[para_into, 195, 80, demod, 5, 223, 81, 206, 81]
582, 581	$n(x+n(x)) + n(x) = n(x).$	[back_demod, 197, demod, 564, 231]
586, 585	$n(n(x)) = x.$	[back_demod, 80, demod, 582]
606, 605	$n(x+n(y)) + n(x+y) = n(x).$	[para_into, 585, 42, flip.1]
621	$A \neq A.$	[back_demod, 10, demod, 606, 586]
622	$\$F.$	[binary, 621, 2]

**KB: Given
to the
system**

----- end of proof -----

PRODUCTION SYSTEMS

1. Forward-chaining Production Systems

- Prolog & other programming languages: rely on backward-chaining
(I.e., given a query, find substitutions that satisfy it)
- Forward-chaining systems: infer everything that can be inferred from KB
each time new sentence is TELL'ed
- Appropriate for agent design: as new percepts come in, forward-chaining returns best action

2. Implementation (Production System)

- One possible approach: use a theorem prover, using resolution to forward-chain over KB
- More restricted systems can be more efficient.
- Typical components:
 - initial KB called "working memory" (positive literals, no variables)
 - rule memory (set of inference rules in form $p_1 \wedge p_2 \wedge \dots \Rightarrow act_1 \wedge act_2 \wedge \dots$)
 - at each cycle: find rules whose premises satisfied by working memory (match phase)
 - decide which should be executed (conflict resolution phase)
 - execute actions of chosen rule (act phase)

3. Match phase

- Unification can do it, but inefficient
- Rete algorithm (used in OPS-5 system): example

rule memory:

$A(x) \wedge B(x) \wedge C(y) \Rightarrow \text{add } D(x)$

$A(x) \wedge B(y) \wedge D(x) \Rightarrow \text{add } E(x)$

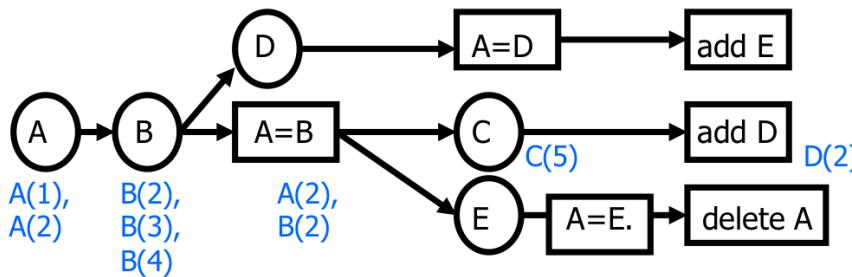
$A(x) \wedge B(x) \wedge E(x) \Rightarrow \text{delete } A(x)$

working memory:

$\{A(1), A(2), B(2), B(3), B(4), C(5)\}$

- Build Rete network from rule memory, then pass working memory through it

ψ. Rete network



Circular nodes: fetches to WM; rectangular nodes: unifications

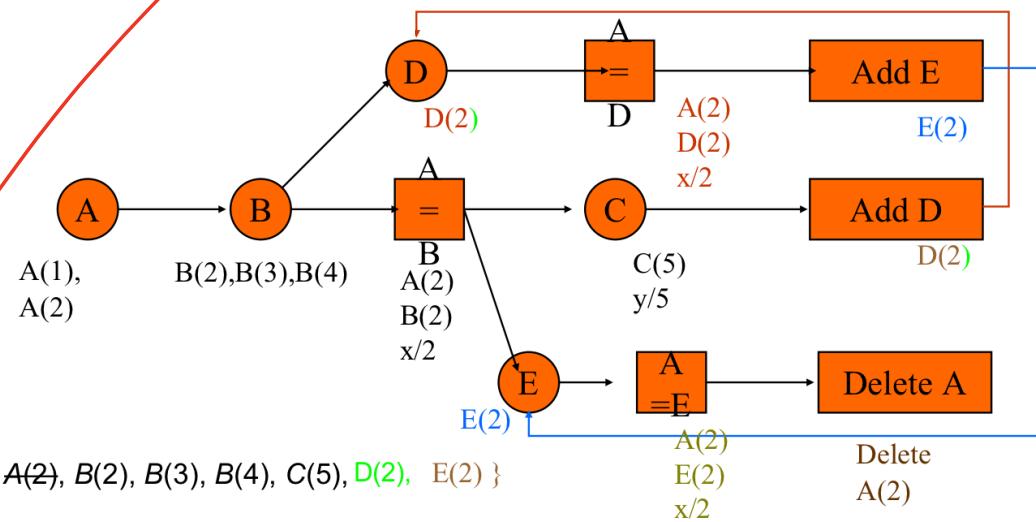
$$A(x) \wedge B(x) \wedge C(y) \Rightarrow \text{add } D(x)$$

$$A(x) \wedge B(y) \wedge D(x) \Rightarrow \text{add } E(x)$$

$$A(x) \wedge B(x) \wedge E(x) \Rightarrow \text{delete } A(x)$$

$$\begin{aligned} A(x) \wedge B(x) \wedge C(y) &\Rightarrow \text{add } D(x) \\ A(x) \wedge B(y) \wedge D(x) &\Rightarrow \text{add } E(x) \\ A(x) \wedge B(x) \wedge E(x) &\Rightarrow \text{delete } A(x) \end{aligned}$$

{A(1), A(2), B(2), B(3), B(4), C(5)}



{ A(1), A(2), B(2), B(3), B(4), C(5), D(2), E(2) }

Delete A(2)

Advantages of Rete networks

- Share common parts of rules
- Eliminate duplication over time (since for most production systems only a few rules change at each time step)

5. Conflict resolution phase

- one strategy: execute all actions for all satisfied rules
- or, treat them as suggestions and use conflict resolution to pick one action.
- Strategies:
 - no duplication (do not execute twice same rule on same args)
 - regency (prefer rules involving recently created WM elements)
 - specificity (prefer more specific rules) When two rules both can be applied, simple one first
 - operation priority (rank actions by priority and pick highest)

三

Frame systems & semantic networks

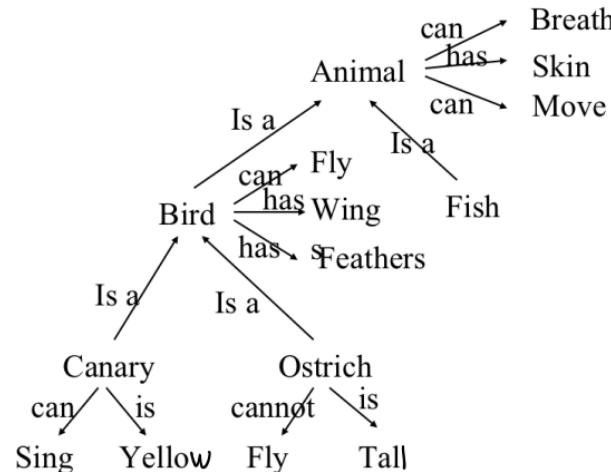
1.

- Other notation for logic; equivalent to sentence notation
- Focus on categories and relations between them (remember ontologies)

- e.g., Cats $\xrightarrow{\text{Subset}}$ Mammals

2.

Syntax and Semantics



Semantic network link types

Link type *New language*

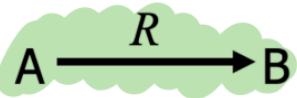


Semantics

$$A \subset B$$



$$A \in B$$



$$R(A, B)$$

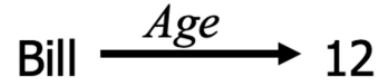
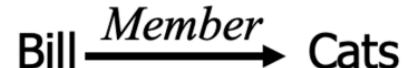


$$\forall x \quad x \in A \Rightarrow R(x, B)$$



$$\forall x \exists y \quad x \in A \Rightarrow y \in B \wedge R(x, y)$$

Example





Description logic System

Goal: Make it easier to describe categories (as opposed to objects)

Popularized by projects such as the Semantic Web

I. DL Syntax

- **Signature**
 - **Concept** (aka **class**) names, e.g., Cat, Animal, Doctor
 - Equivalent to FOL unary predicates
 - **Role** (aka **property**) names, e.g., sits-on, hasParent, loves
 - Equivalent to FOL binary predicates
 - **Individual** names, e.g., Felix, John, Mary, Boston, Italy
 - Equivalent to FOL constants
- **Operators**
 - Many kinds available, e.g.,
 - Standard FOL Boolean operators (\sqcap , \sqcup , \neg)
 - Restricted form of quantifiers (\exists , \forall)
 - Counting (\geq , \leq , $=$)
 - ...

Description logic

(1) concept:

- Concept expressions, e.g.,
 - Doctor \sqcup Lawyer
 - Rich \sqcap Happy
 - Cat \sqcap \exists sits-on.Mat
- Equivalent to FOL formulae with one free variable
 - Doctor(x) \vee Lawyer(x)
 - Rich(x) \wedge Happy(x)
 - $\exists y.(\text{Cat}(x) \wedge \text{sits-on}(x, y))$

- Special concepts

- \top (aka top, Thing, most general concept) meaning True
- \perp (aka bottom, Nothing, inconsistent concept)

used as abbreviations for

- $(A \sqcup \neg A)$ for any concept A
- $(A \sqcap \neg A)$ for any concept A

Description logic

(2) Role:

- Role **expressions**, e.g.,
 - loves
 - hasParent \circ hasBrother
- Equivalent to FOL formulae with two free variables
 - loves(y, x)
 - $\exists z. (\text{hasParent}(x, z) \wedge \text{hasBrother}(z, y))$

(3) Axiom:

① "Schema" **Axioms**, e.g.,

- Rich $\sqsubseteq \neg$ Poor (concept inclusion)
- Cat $\sqcap \exists \text{sits-on}.\text{Mat} \sqsubseteq$ Happy (concept inclusion)
- BlackCat \equiv Cat $\sqcap \exists \text{hasColour}.\text{Black}$ (concept equivalence)
- sits-on \sqsubseteq touches (role inclusion)
- Trans(part-of) (transitivity)
- Equivalent to (particular form of) FOL sentence, e.g.,
 - $\forall x. (\text{Rich}(x) \rightarrow \neg \text{Poor}(x))$
 - $\forall x. (\text{Cat}(x) \wedge \exists y. (\text{sits-on}(x, y) \wedge \text{Mat}(y)) \rightarrow \text{Happy}(x))$
 - $\forall x. (\text{BlackCat}(x) \leftrightarrow (\text{Cat}(x) \wedge \exists y. (\text{hasColour}(x, y) \wedge \text{Black}(y))))$
 - $\forall x, y. (\text{sits-on}(x, y) \rightarrow \text{touches}(x, y))$
 - $\forall x, y, z. ((\text{sits-on}(x, y) \wedge \text{sits-on}(y, z)) \rightarrow \text{sits-on}(x, z))$

Description logic

② “Data” Axioms (aka Assertions or Facts), e.g.,

- BlackCat(Felix) (concept assertion)
- Mat(Mat1) (concept assertion)
- Sits-on(Felix,Mat1) (role assertion)

- Directly equivalent to FOL “ground facts”

- Formulae with no variables

Knowledge base = set of axioms (TBox) + set of facts (ABox)

2. The DL family

- Many different DLs, often with “strange” names
 - E.g., *EL*, *ALC*, *SHIQ*
- Particular DL defined by:
 - Concept operators (\sqcap , \sqcup , \neg , \exists , \forall , etc.)
 - Role operators (\cdot , \circ , etc.)
 - Concept axioms (\sqsubseteq , \equiv , etc.)
 - Role axioms (\sqsubseteq , Trans, etc.)

The DL family

(1) E.g., \mathcal{EL} is a well known “sub-Boolean” DL

- Concept operators: \sqcap , \neg , \exists
 - No role operators (only atomic roles)
 - Concept axioms: \sqsubseteq , \equiv
 - No role axioms
- E.g.:

$\text{Parent} \equiv \text{Person} \sqcap \exists \text{hasChild}.\text{Person}$

(2) \mathcal{ALC} is the smallest propositionally closed DL

- Concept operators: \sqcap , \sqcup , \neg , \exists , \forall
 - No role operators (only atomic roles)
 - Concept axioms: \sqsubseteq , \equiv
 - No role axioms
- E.g.:

$\text{ProudParent} \equiv \text{Person} \sqcap \forall \text{hasChild}.(\text{Doctor} \sqcup \exists \text{hasChild}.\text{Doctor})$

The DL family

- (3) • \mathcal{S} used for \mathcal{ALC} extended with (role) transitivity axioms
 - Additional letters indicate various extensions, e.g.:
 - \mathcal{H} for role hierarchy (e.g., $\text{hasDaughter} \sqsubseteq \text{hasChild}$)
 - \mathcal{R} for role box (e.g., $\text{hasParent} \circ \text{hasBrother} \sqsubseteq \text{hasUncle}$)
 - \mathcal{O} for nominals/singleton classes (e.g., $\{\text{Italy}\}$)
 - \mathcal{I} for inverse roles (e.g., $\text{isChildOf} \equiv \text{hasChild}^{-}$)
 - \mathcal{N} for number restrictions (e.g., $\geq 2\text{hasChild}$, $\leq 3\text{hasChild}$)
 - \mathcal{Q} for qualified number restrictions (e.g., $\geq 2\text{hasChild}.\text{Doctor}$)
 - \mathcal{F} for functional number restrictions (e.g., $\leq 1\text{hasMother}$)
 - E.g., $\mathcal{SHIQ} = \mathcal{S} + \text{role hierarchy} + \text{inverses} + \text{QNRs}$

Example

- W3C's OWL 2 (like OWL, DAML+OIL & OIL) based on DL
 - OWL 2 based on SROIQ, i.e., *ALC* extended with transitive roles, a role box nominals, inverse roles and qualified number restrictions
 - OWL 2 EL based on EL
 - OWL 2 QL based on DL-Lite
 - OWL 2 EL based on DLP
 - OWL was based on SHOIN
 - only simple role hierarchy, and unqualified NRs



Example: OWL2

OWL Constructor	DL Syntax	Example
intersectionOf	$C_1 \sqcap \dots \sqcap C_n$	Human \sqcap Male
unionOf	$C_1 \sqcup \dots \sqcup C_n$	Doctor \sqcup Lawyer
complementOf	$\neg C$	\neg Male
oneOf	$\{x_1\} \sqcup \dots \sqcup \{x_n\}$	{john} \sqcup {mary}
allValuesFrom	$\forall P.C$	\forall hasChild.Doctor
someValuesFrom	$\exists P.C$	\exists hasChild.Lawyer
maxCardinality	$\leq nP$	≤ 1 hasChild
minCardinality	$\geq nP$	≥ 2 hasChild

OWL Syntax	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human \sqsubseteq Animal \sqcap Biped
equivalentClass	$C_1 \equiv C_2$	Man \equiv Human \sqcap Male
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter \sqsubseteq hasChild
equivalentProperty	$P_1 \equiv P_2$	cost \equiv price
transitiveProperty	$P^+ \sqsubseteq P$	ancestor $^+$ \sqsubseteq ancestor

OWL Syntax	DL Syntax	Example
type	$a : C$	John : Happy-Father
property	$\langle a, b \rangle : R$	\langle John, Mary \rangle : has-child

Example: OWL2

OWL RDF/XML Exchange Syntax

E.g., $\text{Person} \sqcap \forall \text{hasChild}.(\text{Doctor} \sqcup \exists \text{hasChild}.\text{Doctor})$:

```
<owl:Class>
  <owl:intersectionOf rdf:parseType=" collection">
    <owl:Class rdf:about="#Person"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasChild"/>
      <owl:allValuesFrom>
        <owl:unionOf rdf:parseType=" collection">
          <owl:Class rdf:about="#Doctor"/>
          <owl:Restriction>
            <owl:onProperty rdf:resource="#hasChild"/>
            <owl:someValuesFrom rdf:resource="#Doctor"/>
          </owl:Restriction>
        </owl:unionOf>
      </owl:allValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

3. Why description logics?

- OWL exploits results of 20+ years of DL research
 - Well defined (model theoretic) **semantics**
 - **Formal properties** well understood (complexity, decidability)



I can't find an efficient algorithm, but neither can all these famous people.

[Garey & Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, 1979.]

- Known **reasoning algorithms**
- **Scalability** demonstrated by **implemented systems**

Why description logics?

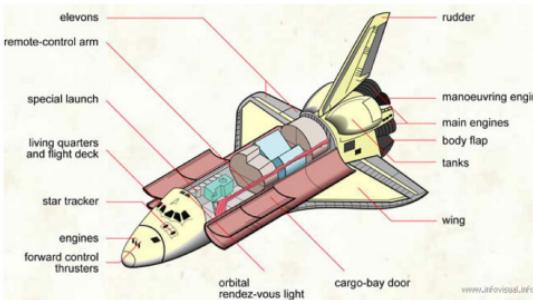
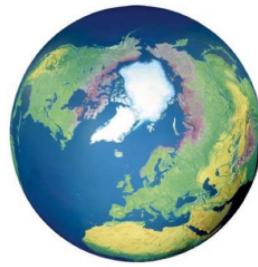
Major benefit of OWL has been huge increase in range and sophistication of tools and infrastructure:

- Editors/development environments
- Reasoners
- Explanation, justification and pinpointing
- Integration and modularisation
- APIs, in particular the [OWL API](#)

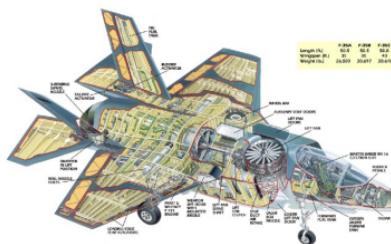
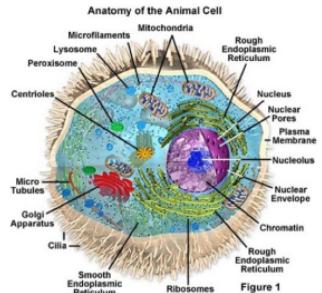
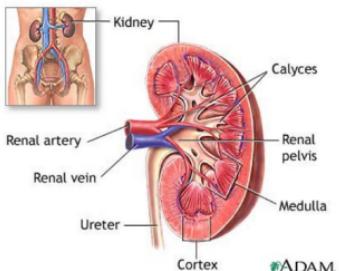
Revision 1403 - ([download](#)) ([annotate](#))
Fri Dec 18 17:14:37 2009 UTC (4 months, 2 weeks ago) by matthewhorridge
File size: 4711 byte(s)

```
1 package org.coode.owlapi.examples;
2
3 import org.semanticweb.owlapi.apibinding.OWLManager;
4 import org.semanticweb.owlapi.model.*;
5 import org.semanticweb.owlapi.util.DefaultPrefixManager;
6 /*
7 * Copyright (C) 2009, University of Manchester
8 *
9 * Modifications to the initial code base are copyright of their
10 * respective authors, or their employers as appropriate. Authorship
11 * of the modifications may be determined from the ChangeLog placed at
12 * the end of this file.
13 *
14 * This library is free software; you can redistribute it and/or
15 * modify it under the terms of the GNU Lesser General Public
16 * License as published by the Free Software Foundation; either
17 * version 2.1 of the License, or (at your option) any later version.
18 *
19 * This library is distributed in the hope that it will be useful,
20 * but WITHOUT ANY WARRANTY; without even the implied warranty of
21 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
22 * Lesser General Public License for more details.
```

Many Applications of Logic Systems



Science (e.g., genomics), geography,



Medicine,

biology,

defense