Logical reasoning systems

• Theorem provers and logic programming languages

• Production systems

• Frame systems and semantic networks

• Description logic systems
Logical reasoning systems

- **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.
Basic tasks

• Add a new fact to KB – TELL

• Given KB and new fact, derive facts implied by conjunction of KB and new fact. In forward chaining: part of TELL

• Decide if query entailed by KB – ASK

• Decide if query explicitly stored in KB – restricted ASK

• Remove sentence from KB: distinguish between correcting false sentence, forgetting useless sentence, or updating KB re. change in the world.
Indexing, retrieval & unification

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

  Compound: has operator & arguments.
  
  \[ \text{e.g., } c = P(x) \land Q(x) \quad \text{Op}[c] = \land; \text{Args}[c] = [P(x), Q(x)] \]

- FETCH: find sentences in KB that have same structure as query. ASK makes multiple calls to FETCH.

- STORE: add each conjunct of sentence to KB. Used by TELL.
  
  e.g., implement KB as list of conjuncts
  
  TELL(KB, A \land \neg B) \quad \text{TELL(KB, } \neg C \land D) \quad \text{then KB contains: } [A, \neg B, \neg C, D]
Complexity

• With previous approach,

  FETCH takes $O(n)$ time on $n$-element KB

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

Faster solution?
Table-based indexing

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

Example:

<table>
<thead>
<tr>
<th>Key</th>
<th>Positive</th>
<th>Negative</th>
<th>Conclusion</th>
<th>Premise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mother</td>
<td>Mother(ann,sam)</td>
<td>-Mother(ann,al)</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td></td>
<td>Mother(grace,joe)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>dog(rover)</td>
<td>-dog(alice)</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td></td>
<td>dog(fido)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table-based indexing

- 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
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
Tree-based indexing

Example:

Person(age, height, weight, income)
Person(30, 72, 210, 45000)
Fetch(Person(age, 72, 210, income))
Fetch(Person(age, height > 72, weight < 210, income))
Unification algorithm: Example

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)
Unification algorithm

• Using clever indexing, can reduce number of calls to unification

• Still, unification called very often (at basis of modus ponens) \(\Rightarrow\) need efficient implementation.

• See AI MA p. 303 for example of algorithm with \(O(n^2)\) complexity
  \((n\ \text{being size of expressions being unified}).\)
Logic programming

Remember: knowledge engineering vs. programming...

Sound bite: computation as inference on logical KBs

<table>
<thead>
<tr>
<th>Logic programming</th>
<th>Ordinary programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify problem</td>
<td>Identify problem</td>
</tr>
<tr>
<td>2. Assemble information</td>
<td>Assemble information</td>
</tr>
<tr>
<td>3. Tea break</td>
<td>Figure out solution</td>
</tr>
<tr>
<td>4. Encode information in KB</td>
<td>Program solution</td>
</tr>
<tr>
<td>5. Encode problem instance as facts</td>
<td>Encode problem instance as data</td>
</tr>
<tr>
<td>6. Ask queries</td>
<td>Apply program to data</td>
</tr>
<tr>
<td>7. Find false facts</td>
<td>Debug procedural errors</td>
</tr>
</tbody>
</table>

Should be easier to debug $Capital(New\, York, \, US)$ than $x := x + 2$!
Logic programming systems

e.g., Prolog:

- 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 ⇒ 10 million LIPS

Program = set of clauses = head :- literal₁, ..., literalₙ.
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., \(X \text{ is } Y*Z+3\)
Closed-world assumption ("negation as failure")
  e.g., not PhD(X) succeeds if PhD(X) fails
Basic syntax of facts, rules and queries

\[
\text{<fact>} ::= \text{<term>} . \\
\text{<rule>} ::= \text{<term>} :- \text{<term>} . \\
\text{<query>} ::= \text{<term>} . \\
\text{<term>} ::= \text{<number>} | \text{<atom>} | \text{<variable>} \\
| \text{<atom>} (\text{<terms>}) \\
\text{<terms>} ::= \text{<term>} | \text{<term>}, \text{<terms>}
\]
A PROLOG Program

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

  ```prolog
  parent(alice, jim).
  parent(jim, tim).
  parent(jim, dave).
  parent(jim, sharon).
  parent(tim, james).
  parent(tim, thomas).
  ```
A PROLOG Program

- 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.
• Now we can ask PROLOG questions:
  
  ?- parent(alice, jim).
  yes
  ?- parent(jim, herbert).
  no
  ?-
• Not very exciting. But what about this:

| ?- 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.
A PROLOG Query II

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

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

Depth-first search from a start state X:

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

No need to loop over S: successor succeeds for each

Appending two lists to produce a third:

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

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

- \texttt{append([], L, L)}
- \texttt{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. \([a, b, c]\) \quad [a, b, c, d, e]
Expanding Prolog

- **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

- **Compilation:** generate built-in theorem prover for different predicates in KB

- **Optimization:** for example through re-ordering
  - e.g., “what is the income of the spouse of the president?”
    \[\text{Income}(s, i) \land \text{Married}(s, p) \land \text{Occupation}(p, \text{President})\]
  - faster if re-ordered as:
    \[\text{Occupation}(p, \text{President}) \land \text{Married}(s, p) \land \text{Income}(s, i)\]
Theorem provers

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

- Organized Techniques for Theorem Proving and Effective Research (McCune, 1992)

- **Set of support (sos):** set of clauses defining facts about 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.
OTTER

• 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 close in sos to usable list, and add to usable list consequences of resolving that close against usable list

• Halt: when refutation found or sos empty
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
- ELP, a prover for equational logic with associative unification
- Automated Reasoning at Argonne

External Work

- Johan Beltanade's Set Theory Work with Otter
- Some other theorem provers
- Otter made for Eincacs (from Holger Schauer)
- GOAL, by Guoyiang 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.
Example: Robbins Algebras Are Boolean

Historical Background

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

\[ x + y = y + x. \quad \text{[commutativity]} \\
(x + y) + z = x + (y + z). \quad \text{[associativity]} \\
n(n(x) + y) + n(n(x) + n(y)) = x. \quad \text{[Huntington equation]} \]

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

\[ n(n(x + y) + n(x + n(y))) = x. \quad \text{[Robbins equation]} \]

- Robbins and Huntington could not find a proof, and the problem was later studied by Tarski and his students.
CS 561, Session 19

Searching ...

Success, in 1.28 seconds!

------------- PROOF --------------

1 \[ n(n(A)+B)+n(n(A)+n(B)) \neq A. \]
2 \[ x=x. \]
3 \[ x+y=y+x. \]
4 \[ (x+y)+z=x+(y+z). \]
5 \[ n(n(x+y)+n(x+n(y))) = x. \]
6 \[ x+x=x. \]
7 \[ \]
8 \[ \]
9 \[ \]
10 \[ n(n(A)+n(B))+n(n(A)+B) \neq A. \]
11 \[ x+(x+y)=x+y. \]
12 \[ x+(y+z)=y+(x+z). \]
13 \[ x+(y+z)=x+(z+y). \]
14 \[ n(n(x)+n(x)+x)=n(x). \]
15 \[ n(n(x)+n(x)+x)=n(x). \]
16 \[ n(n(x)+n(x)+x)=n(x). \]
17 \[ n(n(x)+n(x)+x)=n(x). \]
18 \[ n(n(x)+n(x)+x)=n(x). \]
19 \[ n(n(x)+n(x)+x)=n(x). \]
20 \[ n(n(x)+n(x)+x)=n(x). \]
21 \[ n(n(x)+n(x)+x)=n(x). \]
22 \[ n(n(x)+n(x)+x)=n(x). \]
23 \[ n(n(x)+n(x)+x)=n(x). \]
24 \[ n(n(x)+n(x)+x)=n(x). \]
25 \[ n(n(x)+n(x)+x)=n(x). \]
26 \[ n(n(x)+n(x)+x)=n(x). \]
27 \[ n(n(x)+n(x)+x)=n(x). \]
28 \[ n(n(x)+n(x)+x)=n(x). \]
29 \[ n(n(x)+n(x)+x)=n(x). \]
30 \[ n(n(x)+n(x)+x)=n(x). \]
31 \[ n(n(x)+n(x)+x)=n(x). \]
32 \[ n(n(x)+n(x)+x)=n(x). \]
33 \[ n(n(x)+n(x)+x)=n(x). \]
34 \[ n(n(x)+n(x)+x)=n(x). \]
35 \[ n(n(x)+n(x)+x)=n(x). \]
36 \[ n(n(x)+n(x)+x)=n(x). \]
37 \[ n(n(x)+n(x)+x)=n(x). \]
38 \[ n(n(x)+n(x)+x)=n(x). \]
39 \[ n(n(x)+n(x)+x)=n(x). \]
40 \[ n(n(x)+n(x)+x)=n(x). \]
41 \[ n(n(x)+n(x)+x)=n(x). \]
42 \[ n(n(x)+n(x)+x)=n(x). \]
43 \[ n(n(x)+n(x)+x)=n(x). \]
44 \[ n(n(x)+n(x)+x)=n(x). \]
45 \[ n(n(x)+n(x)+x)=n(x). \]
46 \[ n(n(x)+n(x)+x)=n(x). \]
47 \[ n(n(x)+n(x)+x)=n(x). \]
48 \[ n(n(x)+n(x)+x)=n(x). \]
49 \[ n(n(x)+n(x)+x)=n(x). \]
50 \[ n(n(x)+n(x)+x)=n(x). \]
51 \[ n(n(x)+n(x)+x)=n(x). \]
52 \[ n(n(x)+n(x)+x)=n(x). \]
53 \[ n(n(x)+n(x)+x)=n(x). \]
54 \[ n(n(x)+n(x)+x)=n(x). \]
55 \[ n(n(x)+n(x)+x)=n(x). \]
56 \[ n(n(x)+n(x)+x)=n(x). \]
57 \[ n(n(x)+n(x)+x)=n(x). \]
58 \[ n(n(x)+n(x)+x)=n(x). \]
59 \[ n(n(x)+n(x)+x)=n(x). \]
60 \[ n(n(x)+n(x)+x)=n(x). \]
61 \[ A!=A. \]
62 \[ $F$. \]

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

- One possible approach: use a theorem prover, using resolution to forward-chain over KB

- More restricted systems can be more efficient.

- Typical components:
  - KB called “working memory” (positive literals, no variables)
  - rule memory (set of inference rules in form
    \[ p_1 \land p_2 \land \ldots \Rightarrow \text{act1} \land \text{act2} \land \ldots \]
  - 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)
Match phase

- Unification can do it, but inefficient

- Rete algorithm (used in OPS-5 system): example
  
  **rule memory:**
  
  \[ A(x) \land B(x) \land C(y) \Rightarrow \text{add } D(x) \]
  
  \[ A(x) \land B(y) \land D(x) \Rightarrow \text{add } E(x) \]
  
  \[ A(x) \land B(x) \land E(x) \Rightarrow \text{delete } A(x) \]

  **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
Rete network

Circular nodes: fetches to WM; rectangular nodes: unifications

\[
\begin{align*}
A(x) \land B(x) \land C(y) &\Rightarrow add \ D(x) \\
A(x) \land B(y) \land D(x) &\Rightarrow add \ E(x) \\
A(x) \land B(x) \land E(x) &\Rightarrow delete \ A(x)
\end{align*}
\]

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

\[ A(x) \land B(x) \land C(y) \Rightarrow \text{add } D(x) \]
\[ A(x) \land B(y) \land D(x) \Rightarrow \text{add } E(x) \]
\[ A(x) \land B(x) \land E(x) \Rightarrow \text{delete } A(x) \]

\{ A(1), A(2), B(2), B(3), B(4), C(5), D(2), E(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)
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)
  - operation priority (rank actions by priority and pick highest)
Frame systems & semantic networks

- Other notation for logic; equivalent to sentence notation

- Focus on categories and relations between them (remember ontologies)

  \textit{Subset}

- e.g., Cats \rightarrow Mammals
## Syntax and Semantics

<table>
<thead>
<tr>
<th>Link Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \xrightarrow{\text{Subset}} B$</td>
<td>$A \subset B$</td>
</tr>
<tr>
<td>$A \xrightarrow{\text{Member}} B$</td>
<td>$A \in B$</td>
</tr>
<tr>
<td>$A \xrightarrow{R} B$</td>
<td>$R(A,B)$</td>
</tr>
<tr>
<td>$A \xrightarrow{\neg R} B$</td>
<td>$\forall x \ x \in A \Rightarrow R(x,y)$</td>
</tr>
<tr>
<td>$A \xrightarrow{R} B$</td>
<td>$\forall x \ \exists y \ x \in A \Rightarrow y \in B \land R(x,y)$</td>
</tr>
</tbody>
</table>
Semantic Network Representation

- Animal
  - Is a
    - Bird
      - Is a
        - Canary
          - can
            - Sing
        - Ostrich
          - cannot
            - Fly
      - has
        - Wings
      - Feathers
        - has
          - Fish
        - can
          - Fly
      - has
        - Skin
      - can
        - Move
    - has
      - Breath
  - can
    - Fly
  - is
    - Yellow
    - Tall
## Semantic network link types

<table>
<thead>
<tr>
<th>Link type</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subset</td>
<td>$A \subset B$</td>
<td>Cats $\rightarrow$ Mammals</td>
</tr>
<tr>
<td>Member</td>
<td>$A \in B$</td>
<td>Bill $\rightarrow$ Cats</td>
</tr>
<tr>
<td>$R$</td>
<td>$R(A, B)$</td>
<td>Bill $\rightarrow$ 12</td>
</tr>
<tr>
<td>$R$</td>
<td>$\forall x \ x \in A \Rightarrow R(x, B)$</td>
<td>Birds $\rightarrow$ 2</td>
</tr>
<tr>
<td>$R$</td>
<td>$\forall x \exists y \ x \in A \Rightarrow y \in B \land R(x, y)$</td>
<td>Birds $\rightarrow$ Birds</td>
</tr>
</tbody>
</table>