Automated Reasoning

6

6 Automated Reasoning

A brief history of reasoning

Automated reasoning: reasoning completely automatically by computer programs

450B.C.	Stoics	propositional logic
322B.C.	Aristotle	syllogisms (inference rules), quantifiers
1565	Cardano	probability theory (propositional logic + uncertainty)
1847	Boole	propositional logic (again)
1879	Frege	first-order logic
1922	Wittgenstein	proof by truth tables
1930	Gödel	complete algorithm for FOL
1930	Herbrand	complete algorithm for FOL (reduce to propositional)
1931	Gödel	incomplete algorithm for arithmetic
1960	Davis/Putnam	"practical" algorithm for propositional logic
1965	Robinson	"practical" algorithm for FOL—resolution

Automated theorem proving

Automated theorem proving (ATP): proving (mathematical) theorems by computer programs

Proof methods divide into (roughly) two kinds

Application of inference rules

– Legitimate (sound) generation of new sentences from old

– Proof = a sequence of inference rule applications

Can use inference rules as operators in a standard search alg.

Inference rules include

– forward chaining, backward chaining, resolution

Model checking

truth table enumeration (always exponential in n)

improved backtracking, e.g., DPLL algorithm

heuristic search in model space (sound but incomplete)

e.g., min-conflicts-like hill-climbing algorithms

Proofs

Sound inference: find α such that $KB \vdash \alpha$
Proof process is a search, operators are inference rules

Modus Ponens (MP)

$$\frac{\alpha, \quad \alpha \Rightarrow \beta}{\beta} \qquad \frac{At(lin, pku) \quad At(lin, pku) \Rightarrow Ok(lin)}{Ok(lin)}$$

And-Introduction (AI)

$$\frac{\alpha \quad \beta}{\alpha \wedge \beta} \qquad \frac{Ok(lin) \quad AImajor(lin)}{Ok(Lin) \wedge AImajor(in)}$$

Universal instantiation (UI)

Every instantiation of a universally quantified sentence is entailed by it:

$$\frac{\forall\, v \;\; \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

for any variable v and ground term g

E.g., $\forall\, x \;\; King(x) \land Greedy(x) \Rightarrow Evil(x)$ yields

$King(john) \land Greedy(john) \Rightarrow Evil(john)$
$King(richard) \land Greedy(richard) \Rightarrow Evil(richard)$
$King(father(john)) \land Greedy(father(john)) \Rightarrow Evil(father(john))$
$\vdots$

Existential instantiation (EI)

c For any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base:

$$\frac{\exists\, v \;\; \alpha}{\textsc{Subst}(\{v/k\}, \alpha)}$$

E.g., $\exists\, x \;\; Crown(x) \land OnHead(x, john)$ yields

$$Crown(c) \land OnHead(c, john)$$

provided c is a new constant symbol, called a Skolem constant

Another example: from $\exists\, x \;\; d(x^y)/dy = x^y$ we obtain

$$d(e^y)/dy = e^y$$

provided e is a new constant symbol

Instantiation

UI can be applied several times to **add** new sentences;
the new KB is logically equivalent to the old

EI can be applied once to **replace** the existential sentence;
the new KB is **not** equivalent to the old,
but is satisfiable iff the old KB was satisfiable

Example: a proof

bob is a buffalo	1. $Buffalo(bob)$
pat is a pig	2. $Pig(pat)$
Buffaloes outrun pigs	3. $\forall\, x, y \;\; Buffalo(x) \wedge Pig(y) \Rightarrow Faster(x, y)$
bob outruns pat	$Buffalo(bob) \wedge Pig(pat) \Rightarrow Faster(bob, pat)$
UE 3, $\{x/bob, y/pat\}$	

Example: a proof

AI 1 & 2	4. $Buffalo(bob) \wedge Pig(pat)$

Example: a proof

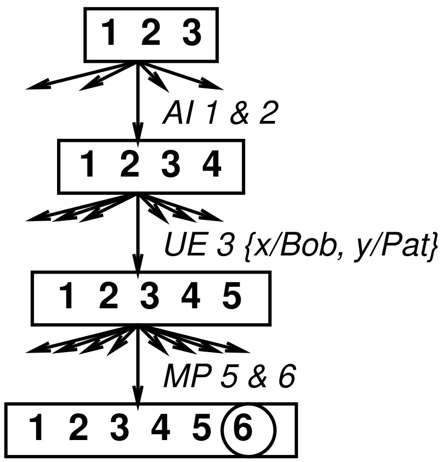
UE 3, $\{x/bob, y/pat\}$	5. $Buffalo(bob) \land Pig(pat) \Rightarrow Faster(bob, pat)$

Example: a proof

MP 6 & 7	6. $Faster(bob, pat)$

Search with inference rules

Operators are inference rules
States are sets of sentences
Goal test checks state to see if it contains a query sentence



AI, UE, MP are common inference patterns

Problem: branching factor huge, esp. for UE

Idea: find a substitution that makes the rule premise match some known facts
$\Rightarrow$ a single, more powerful inference rule

Forward and backward chaining

Modus Ponens (for Horn Form): complete for Horn KBs

$$\frac{\alpha_1, \ldots, \alpha_n, \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

Can be used with forward chaining or backward chaining
These algorithms are very natural and run in **linear** time

Clause form

Clause Form (restricted)

KB = **conjunction** of **clauses** (CNF)

Recall: Clause = disjunction of literals

- proposition symbol; or
- (conjunction of symbols) $\Rightarrow$ symbol
  (i.e., conjunction of literals)

E.g., $C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$
i.e., $C \wedge (\neg B \vee A) \wedge (\neg C \vee \neg D \vee B)$

Horn clause = a clause in which **at most one** is **positive** literal

Definite clause = a clause in which **exactly one** is **positive** literal
all definite clauses are Horn clauses

Goal clauses = clauses with no positive literals

Forward chaining

FC Idea: fire any rule whose premises are satisfied in the KB
add its conclusion to the KB, until query is found

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

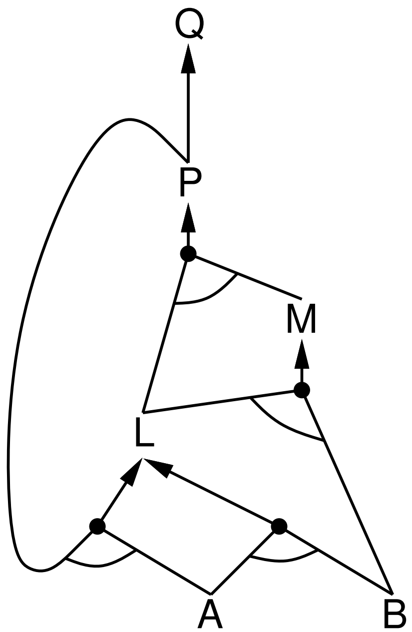
$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

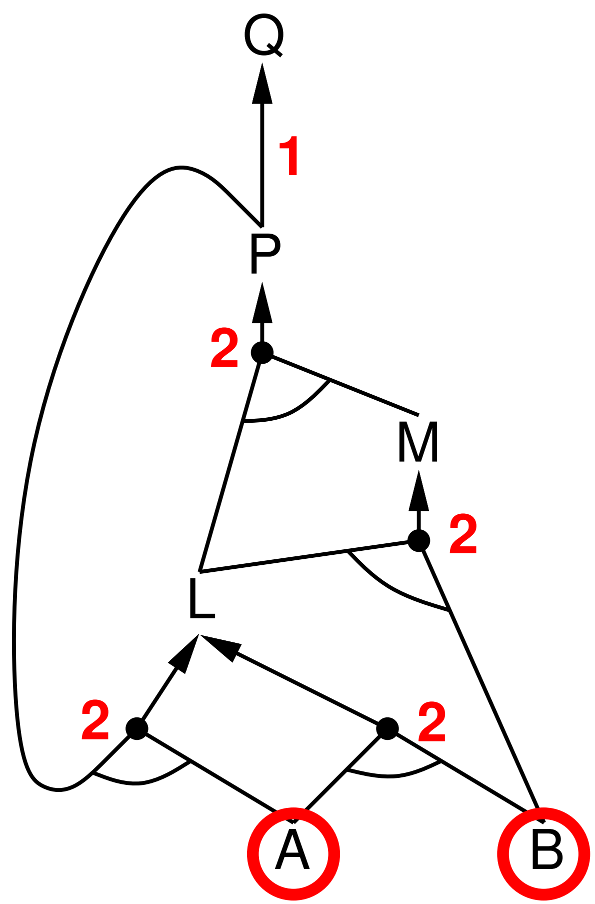
$A \wedge B \Rightarrow L$

A

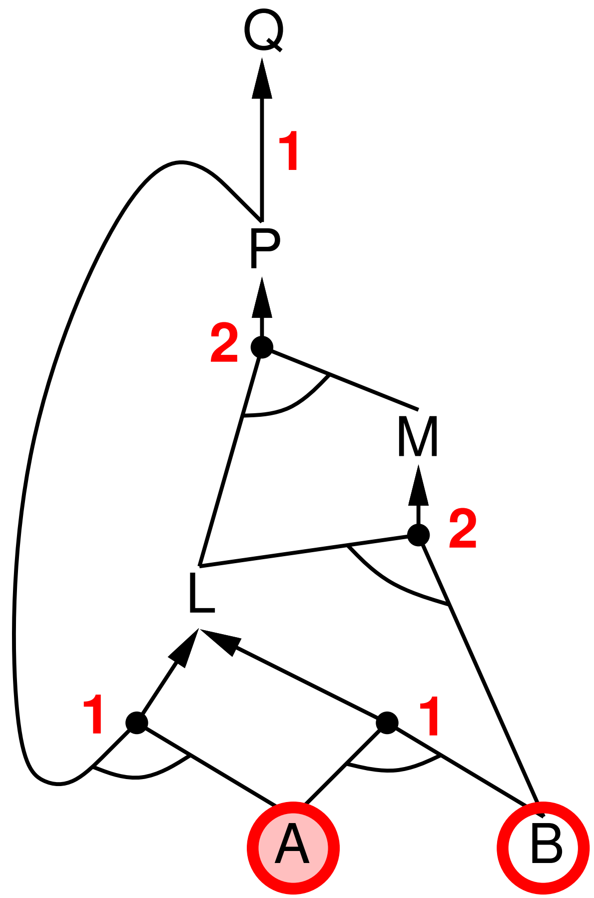
B



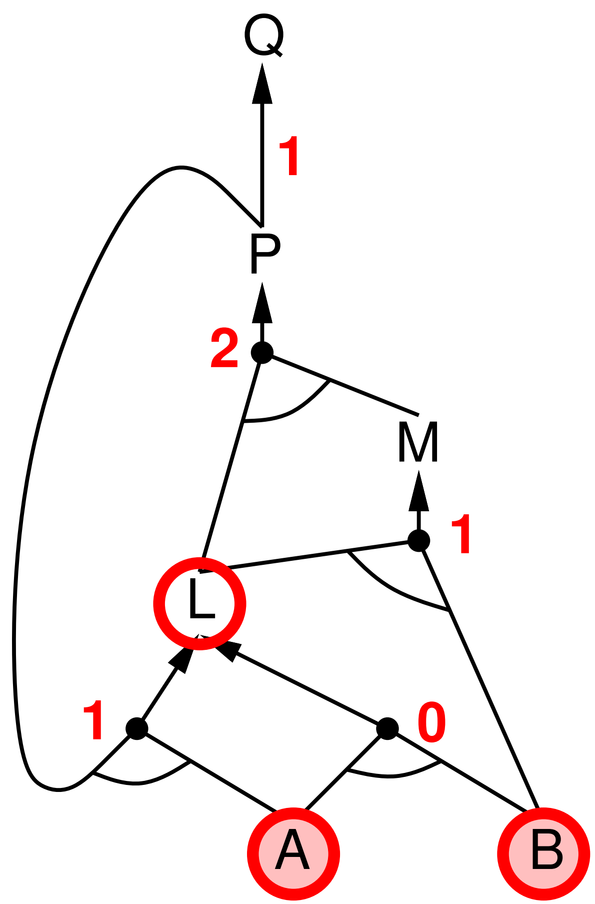
Example: forward chaining



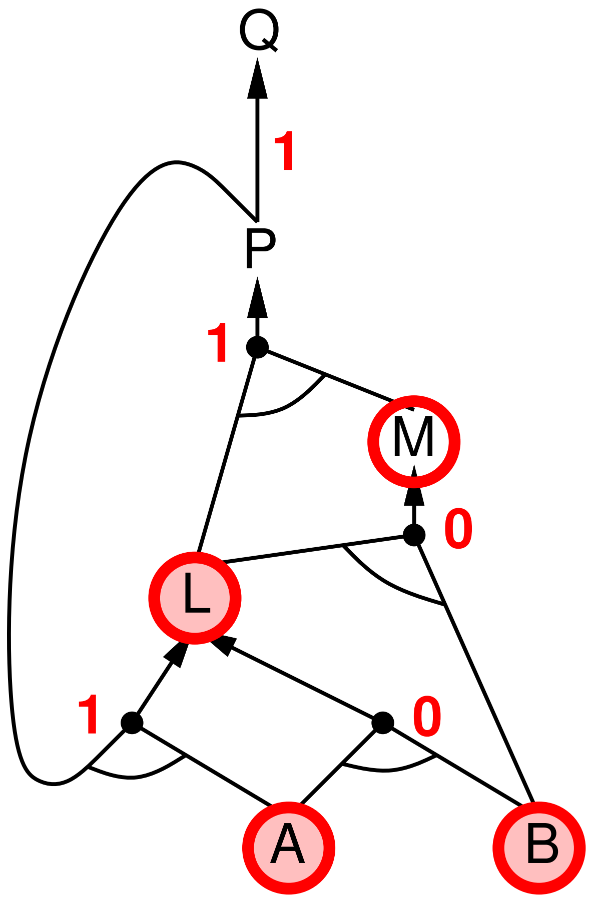
Example: forward chaining



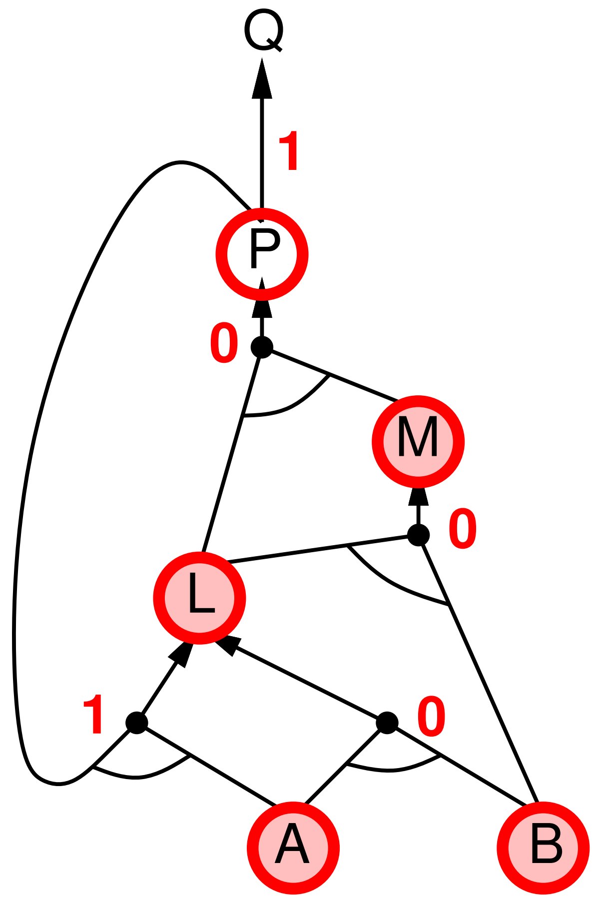
Example: forward chaining



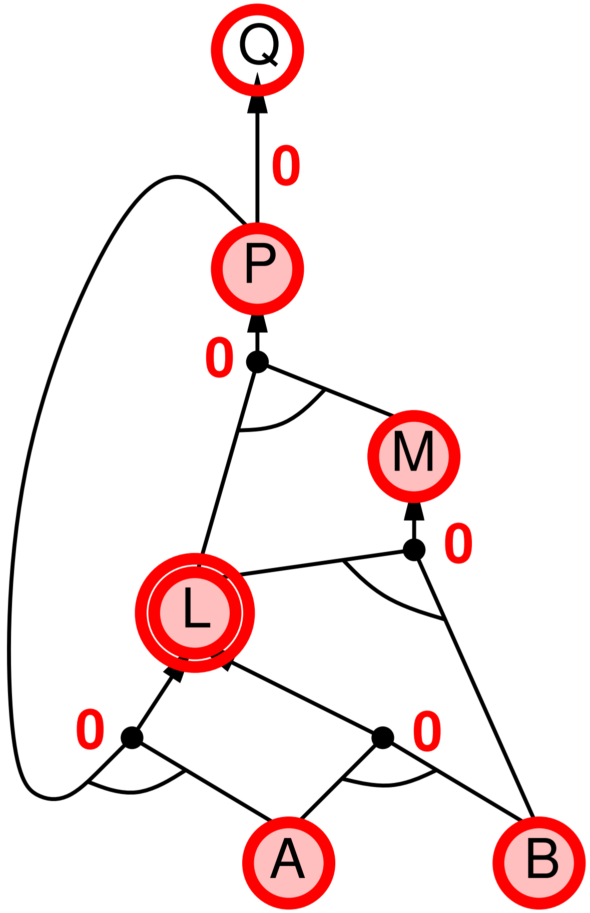
Example: forward chaining



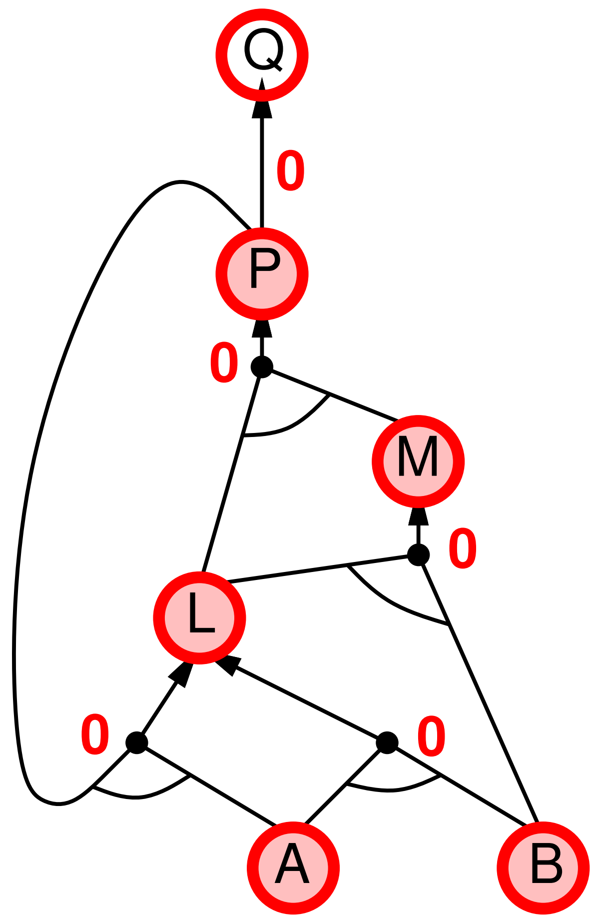
Example: forward chaining



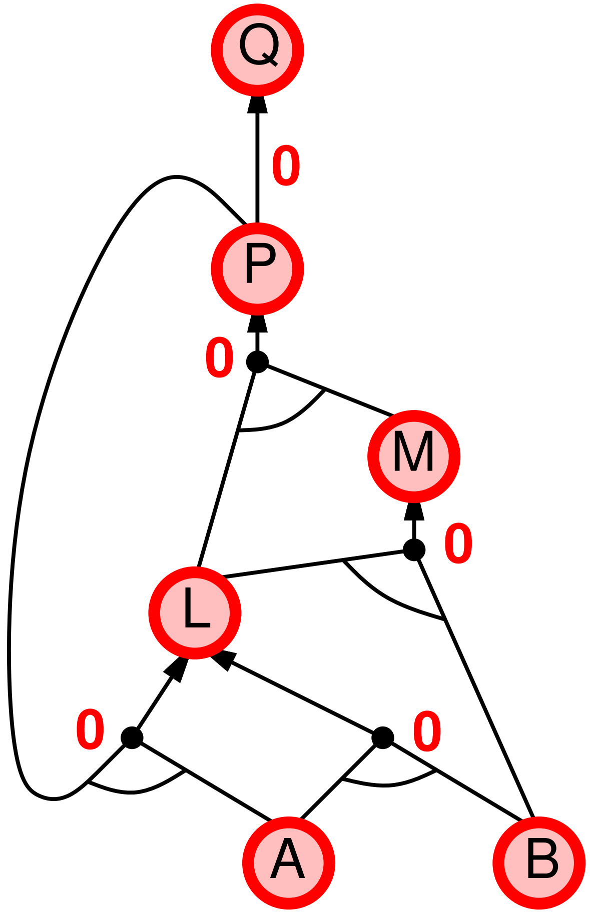
Example: forward chaining



Example: forward chaining



Example: forward chaining



Forward chaining algorithm

def PL-FC-ASK(*KB*, *q*)
    inputs: *KB*, the knowledge base, a set of propositional definite clauses
            q, the query, a proposition symbol
    count ← a table, where *count*[*c*] is the number of symbols in *c*'s premise
    inferred ← a table, where *inferred*[*s*] is initially *false* for all symbols
    queue ← a queue of symbols, initially symbols known to be true in *KB*

    while *queue* is not empty **do** // not yet processed
        p ← POP(*queue*)
        if *p*=*q* **then return** *true*
        if *inferred*[*p*]=*false* **then**
            inferred[*p*] ← *true*
            for each clause *c* in *KB* where *p* is in *c*.PREMISE **do**//implication
                decrement *count*[*c*]
                if *count*[*c*] = 0 **then** add *c*.CONCLUSION to *queue*
    return *false*

Completeness*

FC derives every atomic sentence that is entailed by **Horn** KB

1. FC reaches a fixed point where no new atomic sentences are derived

2. Consider the final state as a model m, assigning true/false to symbols

3. Every clause in the original KB is true in m
   Proof: Suppose a clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in m
   Then $a_1 \wedge \ldots \wedge a_k$ is true in m and b is false in m
   Therefore the algorithm has not reached a fixed point

4. Hence m is a model of KB

5. If $KB \models q$, q is true in **every** model of KB, including m

Idea: construct any model of KB by sound inference, check α

Backward chaining

BC Idea: work backwards from the query q

to prove q by BC

check if q is known already, or

prove by BC all premises of some rule concluding q

Avoid loops: check if new subgoal is already on the goal stack

Avoid repeated work: check if new subgoal

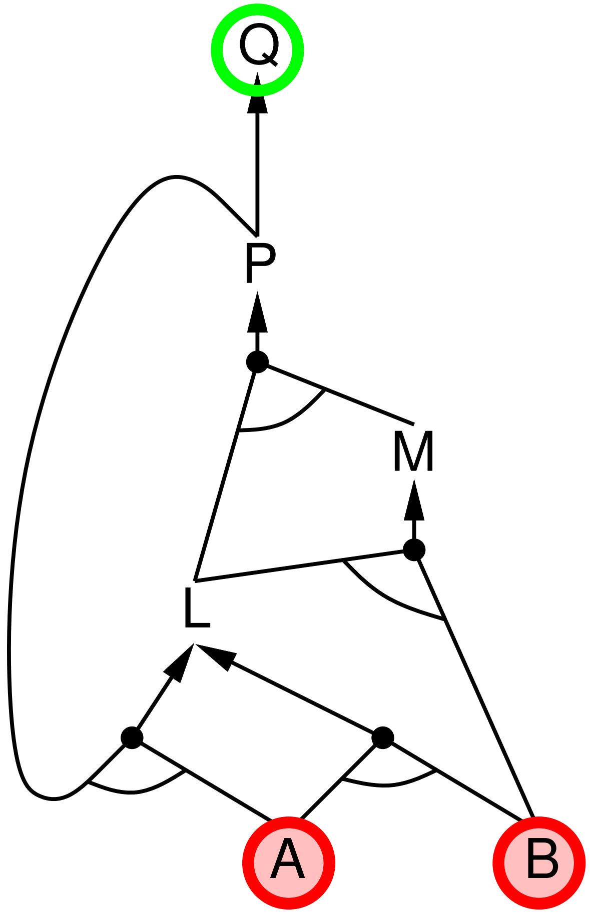
1) has already been proved true, or

2) has already failed

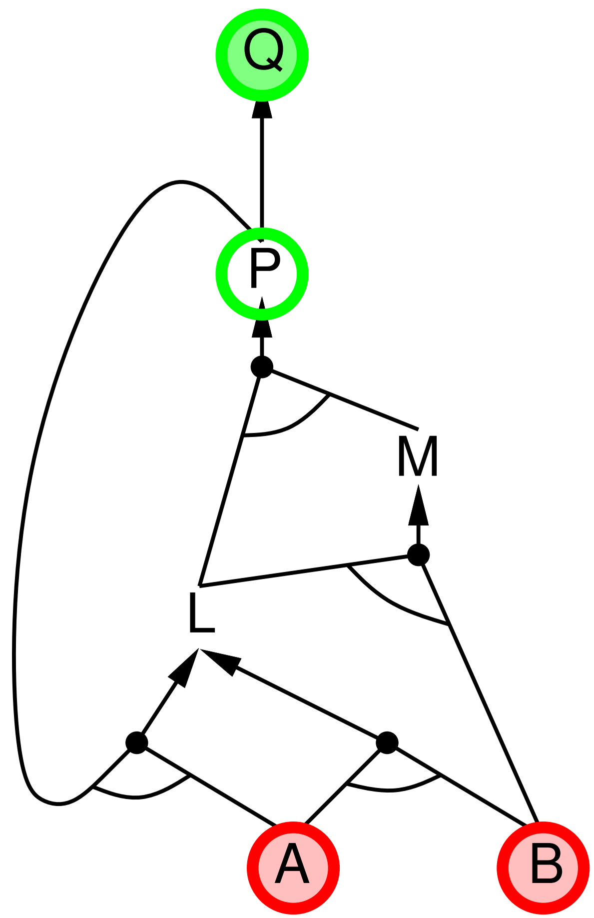
Algorithm: PL-BC-ASK?

(ref. FOL-BC-ASK in later)

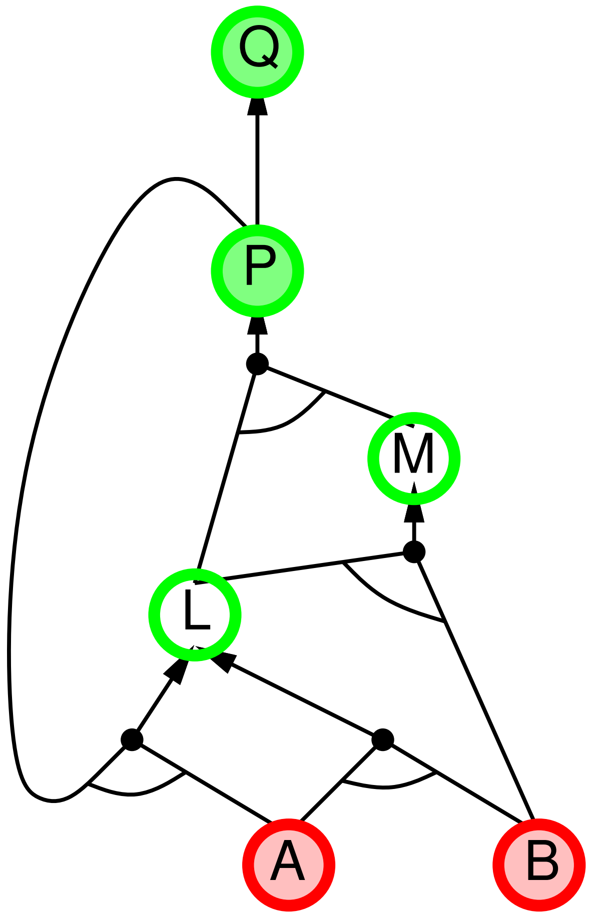
Example: backward chaining



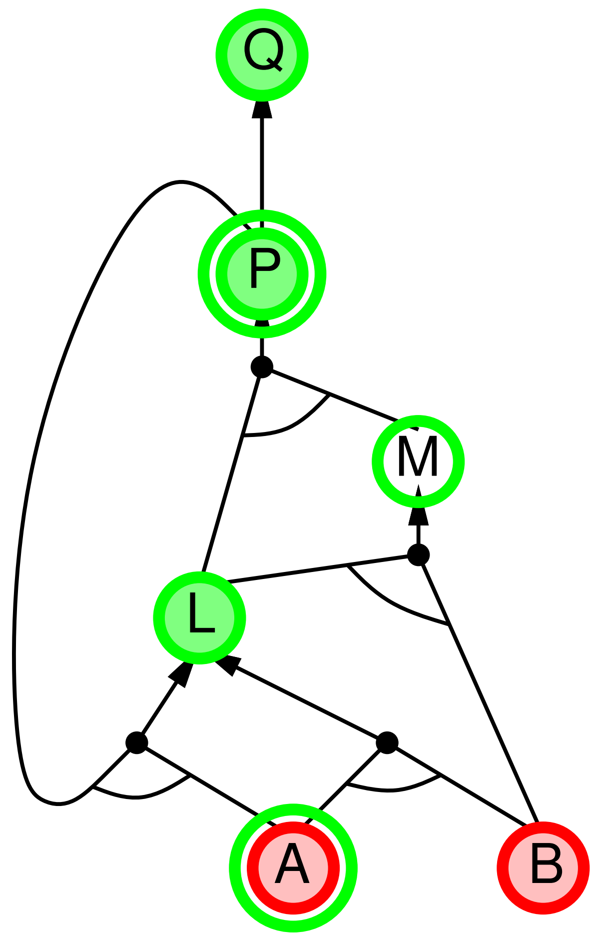
Example: backward chaining



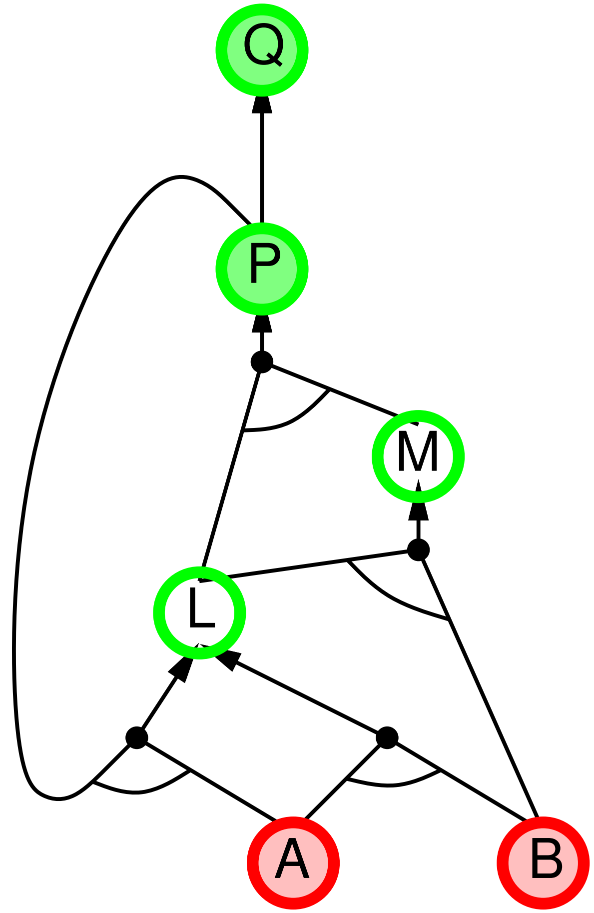
Example: backward chaining



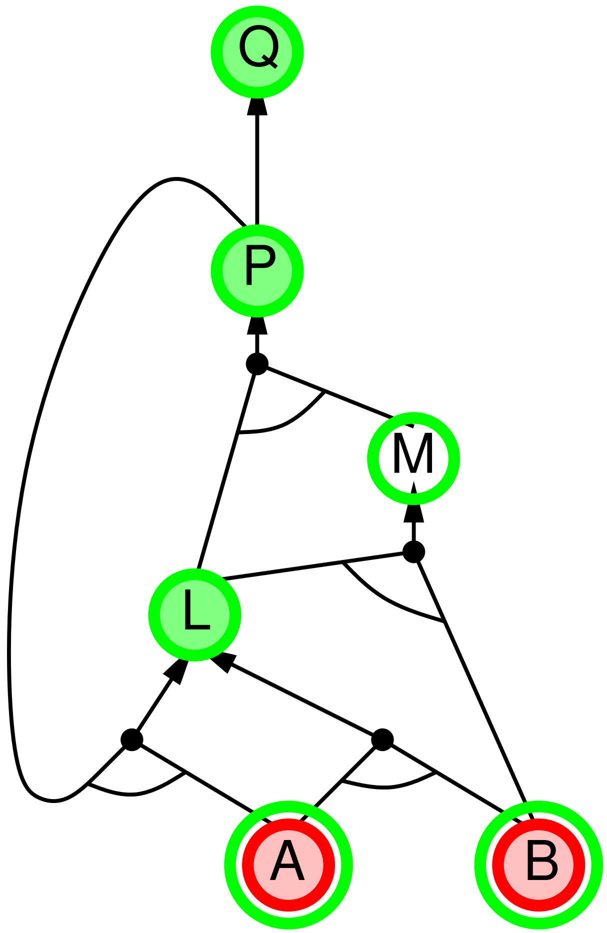
Example: backward chaining



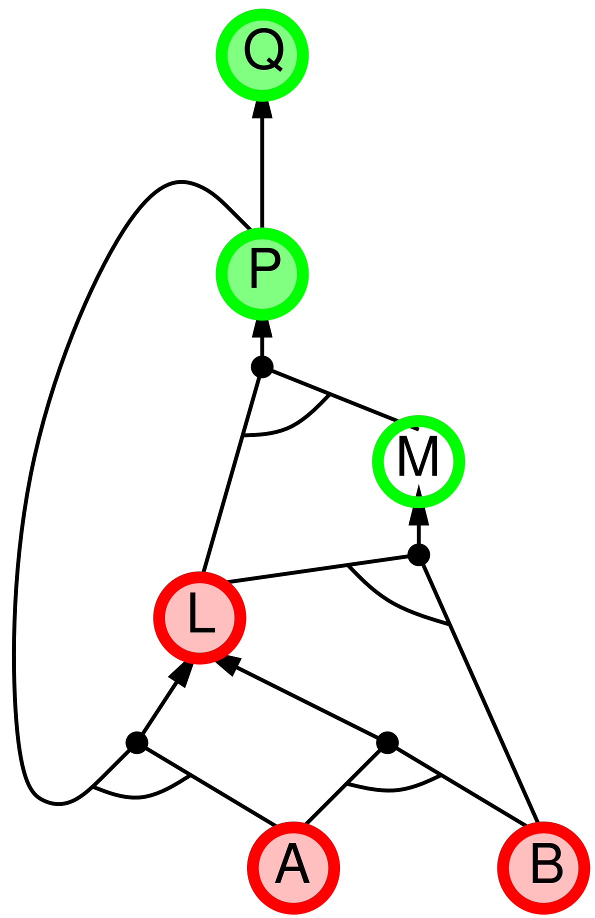
Example: backward chaining



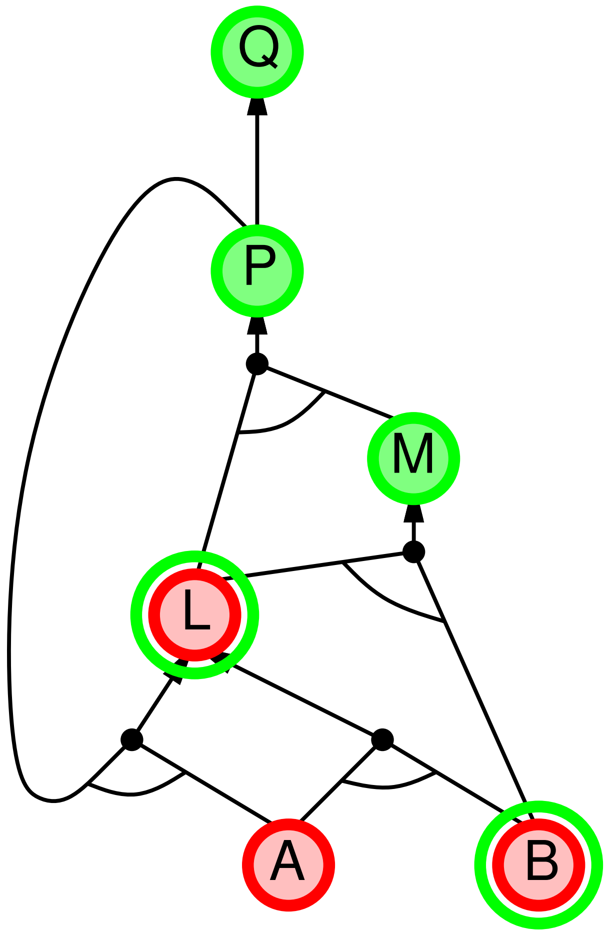
Example: backward chaining



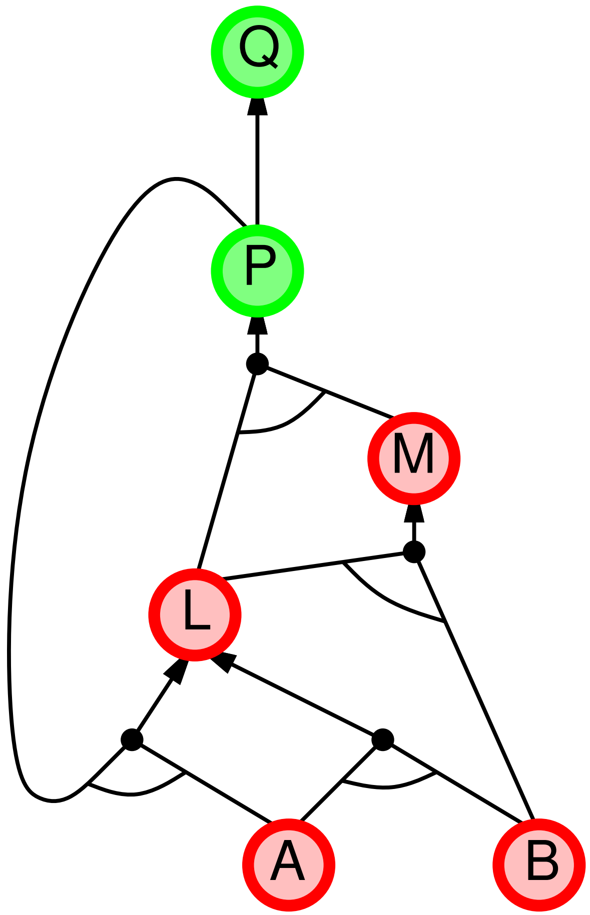
Example: backward chaining



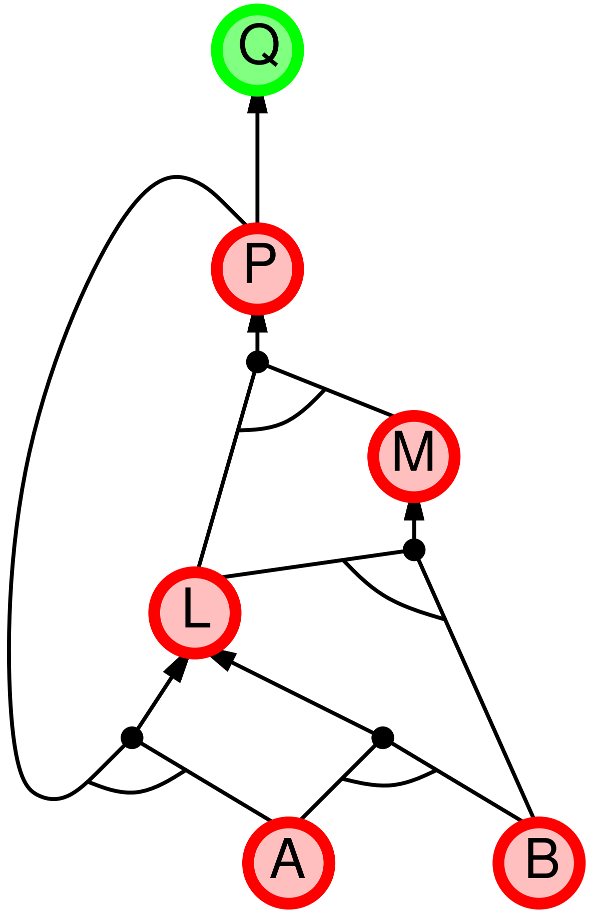
Example: backward chaining



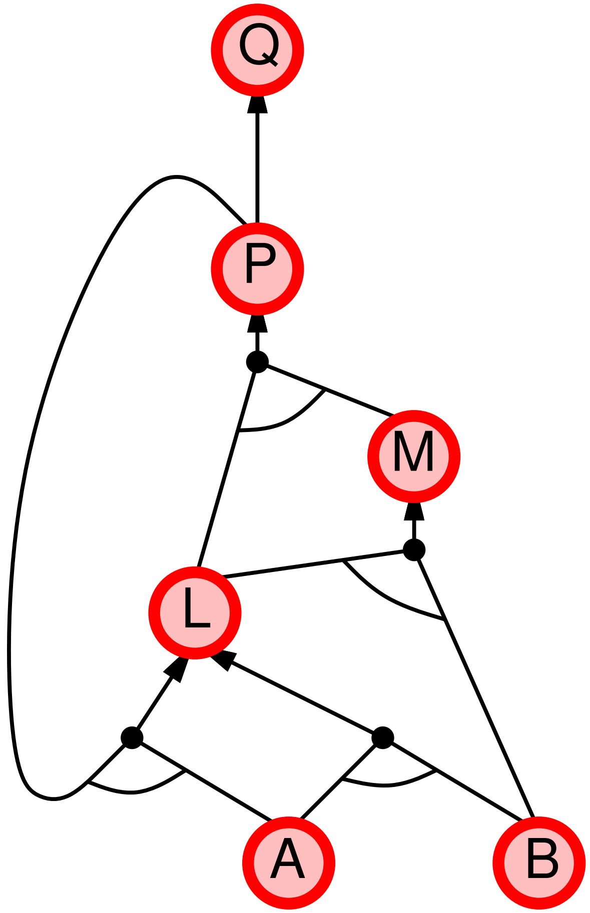
Example: backward chaining



Example: backward chaining



Example: backward chaining



Forward vs. backward chaining

FC is data-driven, cf. automatic, unconscious processing
e.g., object recognition, routine decisions

May do lots of work that is irrelevant to the goal

BC is goal-driven, appropriate for problem-solving
e.g., Where are my keys? How do I get into a PhD program?

Complexity of BC can be **much less** than linear in size of KB

Incompleteness

Forward and backward chaining are **complete for Horn KBs**
but incomplete for full FOL

E.g., from

$$PhD(x) \Rightarrow HighlyQualified(x)$$
$$\neg PhD(x) \Rightarrow EarlyEarnings(x)$$
$$HighlyQualified(x) \Rightarrow Rich(x)$$
$$EarlyEarnings(x) \Rightarrow Rich(x)$$

should be able to infer $Rich(Me)$, but FC/BC won't do it

Does a complete algorithm exist??

Resolution

- Propositional resolution
- Unification
- First-order resolution

Propositional resolution

Entailment in PL is decidable:

can prove that α if $KB \models \alpha$ or$KB \not\models \alpha$

Resolution is a refutation procedure:

to prove $KB \models \alpha$, show that $KB \wedge \neg\alpha$ is unsatisfiable

Resolution uses $KB, \neg\alpha$ in CNF

Resolution inference rule combines two clauses to make a new one

$$\begin{array}{ccc} C_1 & & C_2 \\ & \searrow \quad \swarrow & \\ & C & \end{array}$$

C is called a resolvent of input clauses C_1, C_2

Inference continues until an empty clause $\{\ \}$ is derived (contrad.)

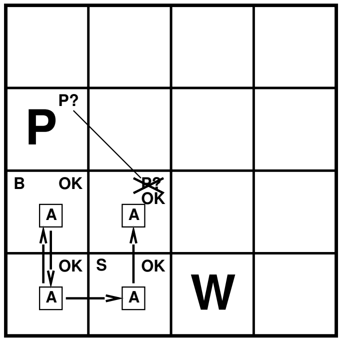
Resolution

Resolution inference rule (for CNF): complete for propositional logic

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

where ℓ_i and m_j are complementary literals. E.g.,

$$\frac{P_{1,3} \vee P_{2,2}, \qquad \neg P_{2,2}}{P_{1,3}}$$



Resolution$^{\#}$

Given a clause of the form $\ell_1 \vee \cdots \vee \ell_k$ containing some literal ℓ_i, and a clause of the form $m_1 \vee \cdots \vee m_n$ containing some literal m_j, where ℓ_i and m_j are complementary literals, infer the clause consisting of those literals in the first clause other than ℓ_i and those in the second other than m_j, i.e.,

$\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$

which is a resolvent of the two input clauses w.r.t. ℓ_i and m_j

A resolution derivation (or proof) of a clause c from a set of clauses S is a sequence of clauses $c_1, \cdots, c_n$, where the last clause, c_n, is c, and where each c_i is either an element of S or a resolvent of two earlier clauses in the derivation

write $S \vdash_i c$ (i is resolution, hereafter simply $\vdash$)

if there is a derivation of c from S

write $\{\,\} \vdash c$, simply $\vdash c$, called c is a theorem

Conversion to CNF

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. Move $\neg$ inwards using de Morgan's rules and double-negation

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

4. Apply distributivity law ($\vee$ over $\wedge$) and flatten

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

Resolution algorithm

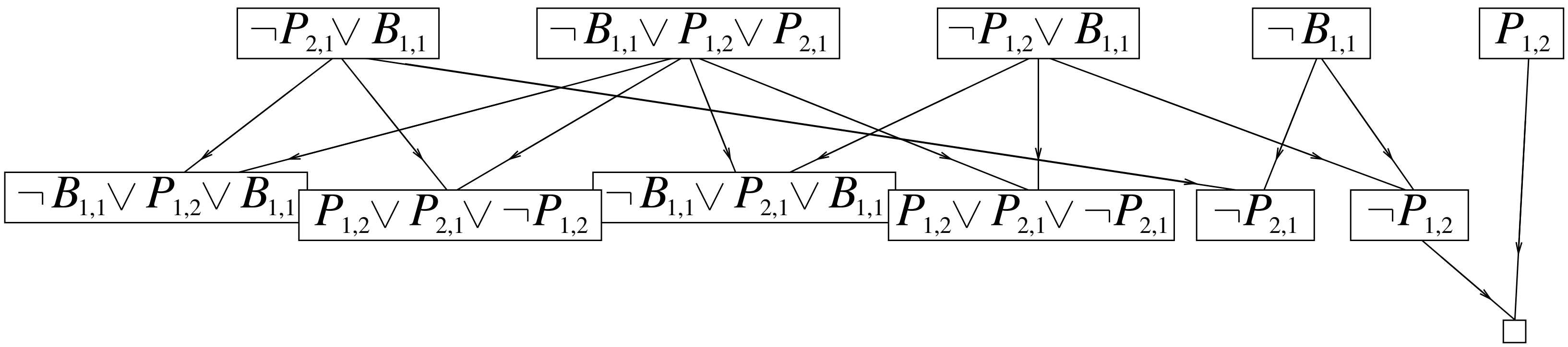
Proof by contradiction, i.e., show $KB \wedge \neg\alpha$ unsatisfiable

```
def PL-Resolution(KB, α)
    inputs: KB, the knowledge base, a sentence in propositional logic
            α, the query, a sentence in propositional logic

    clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
    new ← { }
    while true do
        for each C_i, C_j in clauses do
            resolvents ← PL-Resolve(C_i, C_j)
            if resolvents contains the empty clause then return true
            new ← new ∪ resolvents
        if new ⊆ clauses then return false //unsatisfiable
        clauses ← clauses ∪ new
```

Example: resolution

$KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$ $\alpha = \neg P_{1,2}$



Note: need only convert KB to CNF once

- can handle multiple queries with same KB
- after addition of new fact α, can simply add new clauses α' to KB

Derivation and entailment*

Claim: resolvent is entailed by input clauses

Proof: Suppose $m \models p \vee \alpha$ and $m \models \neg p \vee \beta$

Case 1: $m \models p$

then $m \models \beta$, so $m \models (\alpha \vee \beta)$

Case 2: $m \not\models p$

then $m \models \beta$, so $m \models (\alpha \vee \beta)$

Either way, $m \models (\alpha \vee \beta)$

$$\{(p \vee \alpha), (\neg p \vee \beta)\} \models (\alpha \vee \beta)$$

Special case: c and $\neg c$ resolve to $\{\ \}$

i.e., $\{c, \neg c\}$ is unsatisfiable

Derivation and entailment*

Can extend the previous argument to derivations

If $KB \vdash c$ then $KB \models c$
Proof: by induction on the length of the derivation
Show (by looking at the two cases) that $KB \models c_i$
But the converse does not hold in general

Can have $KB \models c$ without having $KB \vdash c$
E.g., $\neg p \models \neg p \vee \neg q$
but no derivation

Note: resolution is sound but not complete in general

Soundness and completeness of resolution

Theorem: i (resolution) is sound and **refutation** complete if

$KB \vdash_i \alpha$ iff $KB \models \alpha$

A set of clauses is unsatisfiable iff
the resolution closure of those clauses contains the empty clause
– provides method for determining satisfiability: search all derivations for $\{\,\}$
– so provides a method for determining all entailments

Proof of soundness
– Consider the complementary literals ℓ_i,m_j, easy to check

Completeness*

Resolution closure $RC(S)$ (of a set of clauses S) denotes the set of all clauses derivable by resolution; $RC(S)$ must be finite

1. Consider the contrapositive: if the closure $RC(S)$ does not contains the empty clause, then S is satisfiable

2. Construct a model for S with suitable truth values for the symbols $P_1, \cdots, P_k$ that appear in S:

For i from 1 to k

– If a clause in $RC(S)$ contains $\neg P_i$ and all its other literals are false under the assignment chosen for $P_1, \cdots, P_{i-1}$, then assign $false$ to P_i

– Otherwise, assign $true$ to P_i

3. This assignment to $P_1, \cdots, P_k$ is a model of S

Proof by contradiction: at some stage i in the sequence, assigning symbol P_i causes some clause C to become false

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(john)$ and $Greedy(y)$

$\theta = \{x/john, y/john\}$ works

$\textsc{Unify}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(john, x)$	$Knows(john, jane)$	
$Knows(john, x)$	$Knows(y, lin)$	
$Knows(john, x)$	$Knows(y, mother(y))$	
$Knows(john, x)$	$Knows(x, lin)$	

Unification

We can get the inference immediately if we can find a substitution θ s.t. $King(x)$ and $Greedy(x)$ match $King(john)$ and $Greedy(y)$

$\theta = \{x/john, y/john\}$ works

$\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(john, x)$	$Knows(john, jane)$	$\{x/jane\}$
$Knows(john, x)$	$Knows(y, lin)$	
$Knows(john, x)$	$Knows(y, mother(y))$	
$Knows(john, x)$	$Knows(x, lin)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(john)$ and $Greedy(y)$

$\theta = \{x/john, y/john\}$ works

$\textsc{Unify}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(john, x)$	$Knows(john, jane)$	$\{x/jane\}$
$Knows(john, x)$	$Knows(y, lin)$	$\{x/lin, y/john\}$
$Knows(john, x)$	$Knows(y, mother(y))$	
$Knows(john, x)$	$Knows(x, lin)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(john)$ and $Greedy(y)$

$\theta = \{x/john, y/john\}$ works

$\textsc{Unify}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(john, x)$	$Knows(john, jane)$	$\{x/jane\}$
$Knows(john, x)$	$Knows(y, lin)$	$\{x/lin, y/john\}$
$Knows(john, x)$	$Knows(y, mother(y))$	$\{y/john, x/mother(john)\}$
$Knows(john, x)$	$Knows(x, lin)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(john)$ and $Greedy(y)$

$\theta = \{x/john, y/john\}$ works

$\textsc{Unify}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(john, x)$	$Knows(john, jane)$	$\{x/jane\}$
$Knows(john, x)$	$Knows(y, lin)$	$\{x/lin, y/john\}$
$Knows(john, x)$	$Knows(y, mother(y))$	$\{y/john, x/mother(john)\}$
$Knows(john, x)$	$Knows(x, lin)$	$fail$

Standardizing apart eliminates overlap of variables, e.g., $Knows(z, lin)$

Most general unifiers

θ is a most general unifier (MGU, written as $\textsc{Unify}$) of literals l_1 and l_2 iff

1. θ unifies l_1 and l_2
2. for any other unifier θ', there is a another substitution θ^* s.t.
   $\theta' = \theta\theta^*$

   where $\theta\theta^*$ requires applying θ^* to terms in θ

E.g., $P(g(x), f(x), z)$, $\neg P(y, f(w), a)$
  an MGU is
  θ={x/w, y/g(w), z/a}

Theorem: Can limit search to most general unifiers only without loss of completeness

There is a better **linear** algorithm

Algorithm of computing MGUs

Given a set of literals $\{l_i\}$ (usually only two literals)

1. Start with $\theta := \{\}$.
2. If all the $\alpha\theta$ are identical, then done;
   otherwise, get disagreement set, DS
   e.g $P(a, f(a, g(z)))$, $P(a, f(a, u))$, $DS = \{u, g(z)\}$
3. Find a variable $v \in DS$, and a term $t \in DS$ not containing v;
   If not, fail.
4. $\theta := \theta\{v/t\}$
5. Go to 2

There is a better *linear* algorithm

Generalized Modus Ponens (GMP)

$$\frac{p_1{}',\ \ p_2{}',\ \ldots,\ p_n{}',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q\theta}$$ where $p_i{}'\theta = p_i\theta$ for all i

$p_1{}'$ is $King(john)$	p_1 is $King(x)$
$p_2{}'$ is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/john, y/john\}$	q is $Evil(x)$
$q\theta$ is $Evil(john)$	

GMP used with KB of definite clauses (**exactly** one positive literal)
All variables assumed universally quantified

Note: Need to replace all variables in its arguments of a rule with new ones that have not been used before
(variable renaming, STANDARDIZE-VARIABLES function).

Hint: Special interesting for rule-based systems

Soundness of GMP*

Need to show that

$$p_1{}',\ \ldots,\ p_n{}',\ \ (p_1 \wedge \ldots \wedge p_n \Rightarrow q) \models q\theta$$

provided that $p_i{}'\theta = p_i\theta$ for all i

Lemma: For any definite clause p, we have $p \models p\theta$ by UI

1. $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \models (p_1 \wedge \ldots \wedge p_n \Rightarrow q)\theta = (p_1\theta \wedge \ldots \wedge p_n\theta \Rightarrow q\theta)$

2. $p_1{}',\ \ldots,\ p_n{}' \models p_1{}' \wedge \ldots \wedge p_n{}' \models p_1{}'\theta \wedge \ldots \wedge p_n{}'\theta$

3. From 1 and 2, $q\theta$ follows by ordinary Modus Ponens

Example: a small KB

... it is a crime for an American to sell weapons to hostile nations
$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$

Nono ... has some missiles, i.e., $\exists\, x\; Owns(Nono, x) \wedge Missile(x)$
$Owns(Nono, M_1)$ and $Missile(M_1)$

... all of its missiles were sold to it by Colonel West
$\forall\, x \;\; Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

Missiles are weapons
$Missile(x) \Rightarrow Weapon(x)$

An enemy of America counts as "hostile"
$Enemy(x, America) \Rightarrow Hostile(x)$

West, who is American ...
$American(West)$

The country Nono, an enemy of America ...
$Enemy(Nono, America)$

Forward and backward chaining

Recall FC and BC in propositional level, and extend to first-order case

FC is data-driven

BC is goal-oriented

the basis for logic programming, e.g., Prolog

(More complications help to avoid infinite loops)

Two chainings: find **any** solution, find **all** solutions

Forward chaining algorithm#

def FOL-FC-ASK(KB, α)
  inputs: KB, a set of first-order definite clauses
    α, the query (an atomic sentence)

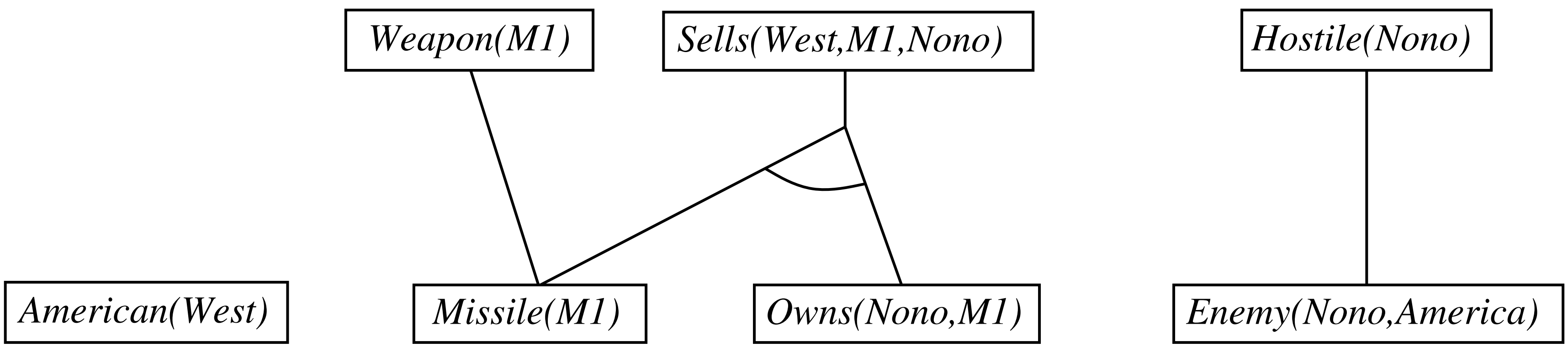
  while *true* **do**
  $new \leftarrow \{\}$ // The set of new sentences inferred on each iteration
  for each *rule* **in** KB **do**
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
    for each θ s.t. SUBST$(\theta, p_1 \wedge \ldots \wedge p_n)$=SUBST$(\theta, p'_1 \wedge \ldots \wedge p'_n)$
      for some $p'_1, \ldots, p'_n$ in KB
      $q' \leftarrow$ SUBST(θ, q)
      if q' does not unify with some sentence already in KB or *new* **then**
        add q' to *new*
        $\theta \leftarrow$ UNIFY(q', α)
        if θ is not *failure* **then return** θ
    if $new = \{\}$ **then return**to *false*
    add *new* to KB

Forward chaining proof

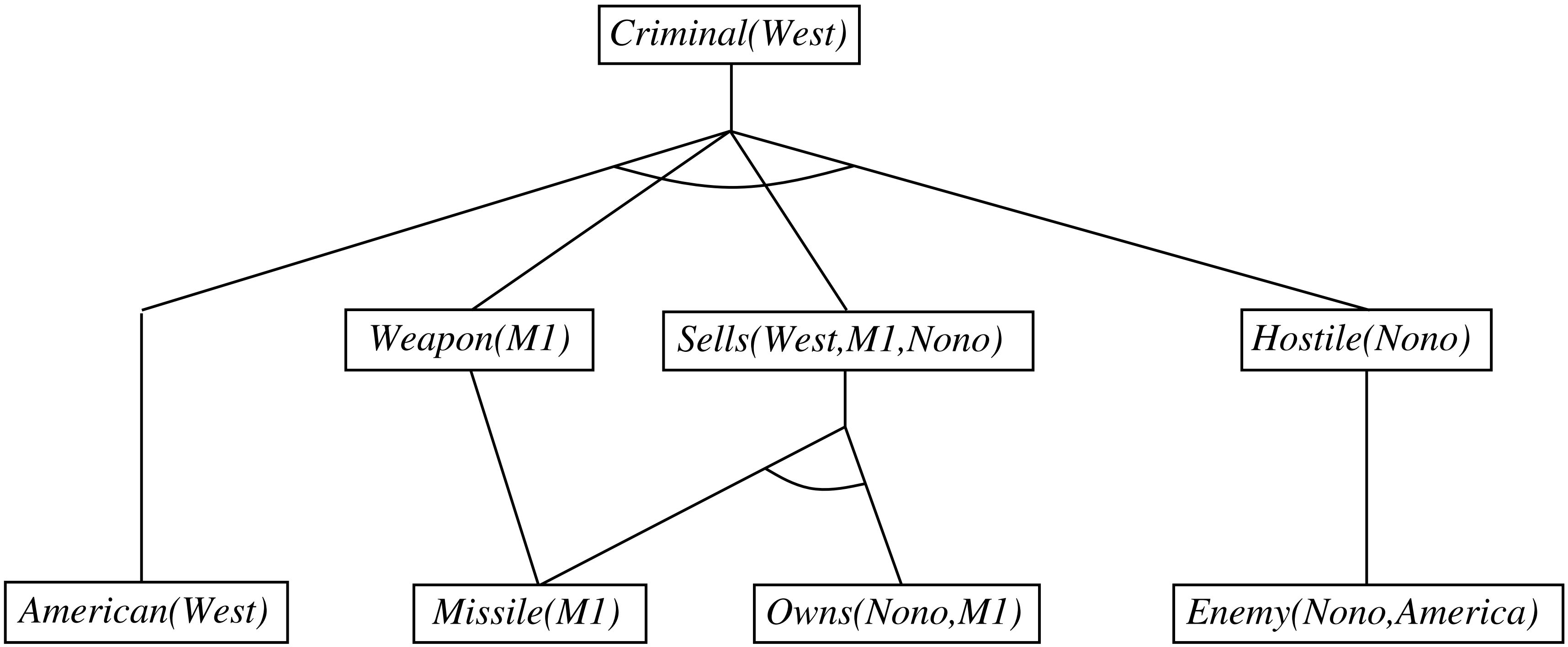
American(West) *Missile(M1)* *Owns(Nono,M1)* *Enemy(Nono,America)*

Hint: can you notice that FOL-FC-ASK differs from PL-FC-ENTAIL?

Forward chaining proof



Forward chaining proof



Properties of forward chaining

Sound and complete for first-order definite clauses
(proof similar to propositional proof)

Datalog = first-order definite clauses + **no functions** (e.g., crime KB)
FC terminates for Datalog in poly iterations: at most $p \cdot n^k$ literals

Logica (logic+aggregation, Google 2021) compiles to SQL and run on Google BigQuery

May not terminate in general if α is not entailed

This is unavoidable: entailment with definite clauses is semidecidable

Efficiency of forward chaining

Simple observation: no need to match a rule on iteration k
if a premise wasn't added on iteration $k-1$
$\Rightarrow$ match each rule whose premise contains a newly added literal

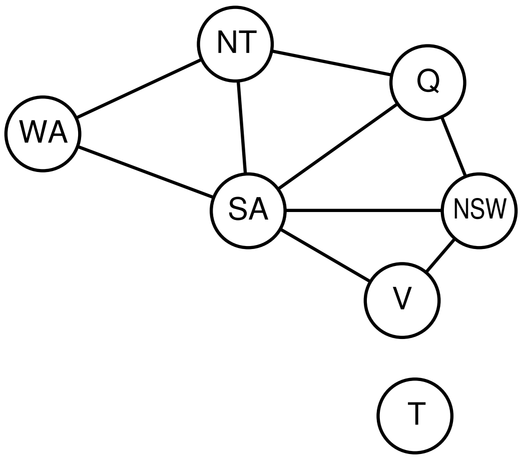
Matching itself can be expensive

Database indexing allows $O(1)$ retrieval of known facts
e.g., query $Missile(x)$ retrieves $Missile(M_1)$

Matching conjunctive premises against known facts is NP-hard

Forward chaining is widely used in deductive databases

Hard matching example



$$
\begin{aligned}
&\mathit{Diff}(wa, nt) \land \mathit{Diff}(wa, sa) \land \\
&\qquad \mathit{Diff}(nt, q) \mathit{Diff}(nt, sa) \land \\
&\qquad \mathit{Diff}(q, nsw) \land \mathit{Diff}(q, sa) \land \\
&\qquad \mathit{Diff}(nsw, v) \land \mathit{Diff}(nsw, sa) \land \\
&\qquad \mathit{Diff}(v, sa) \Rightarrow Colorable()
\end{aligned}
$$

$\mathit{Diff}(Red, Blue)$ $\mathit{Diff}(Red, Green)$
$\mathit{Diff}(Green, Red)$ $\mathit{Diff}(Green, Blue)$
$\mathit{Diff}(Blue, Red)$ $\mathit{Diff}(Blue, Green)$

$Colorable()$ is inferred iff the CSP has a solution
CSPs include 3SAT as a special case, hence matching is NP-hard

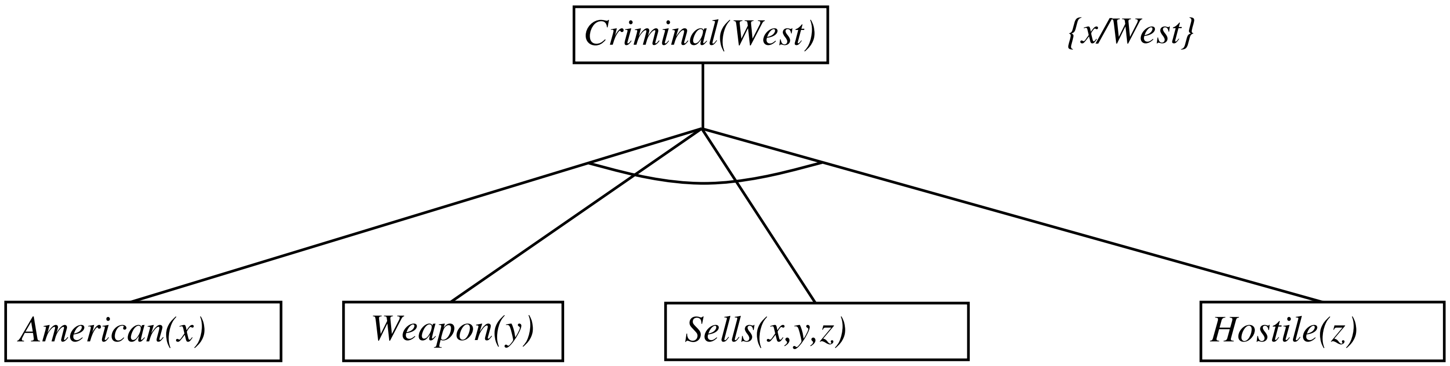
Backward chaining algorithm#

```
def FOL-BC-Ask(KB, query)
   return FOL-BC-Or(KB, query, {}) //And-Or search
def FOL-BC-Or(KB, goal, θ) // Or because querying goal by any rule
   for each rule in Fetch-Rules-For-Goal(KB,goal) do
      (lhs ⇒ rhs) ← Standardize-Variables(rule)
      for each θ′ in FOL-BC-And(KB, lhs, Unify(rhs, goal, θ)) do
         yield θ′ // return by a generator for multiple substitutions
def FOL-BC-And(KB, goal, θ) // And because lhs is a list of conjuncts
   if θ = failure then return
   else if Length(goal) = 0 then yield θ
   else
      first,rest ← First(goal),Rest(goal)
      for each θ′ in FOL-BC-Or(KB, Subst(θ, first), θ) do
         for each θ″ in FOL-BC-And(KB, rest), θ′) do
            yield θ″
```

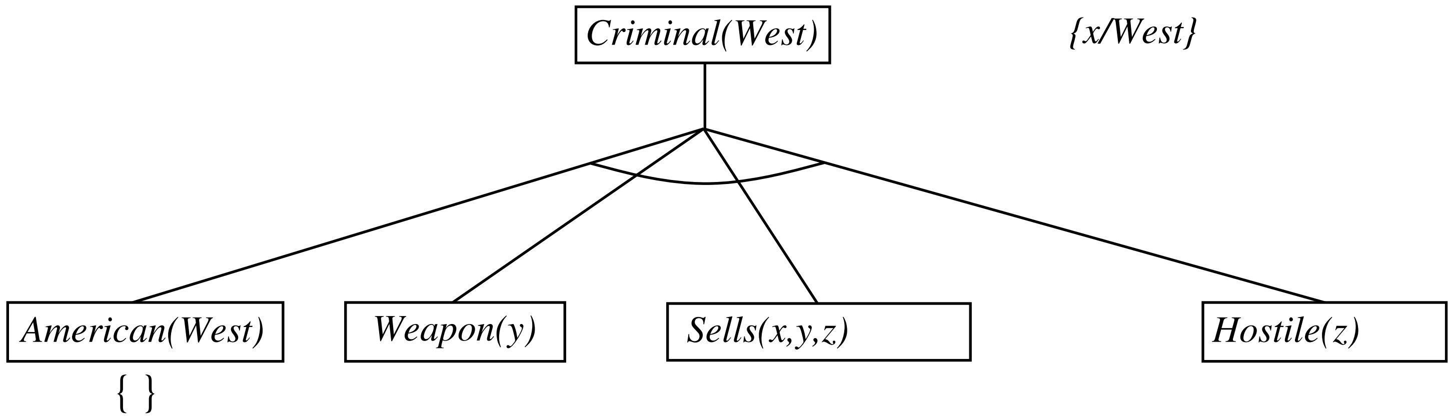
Example: backward chaining

$\boxed{Criminal(West)}$

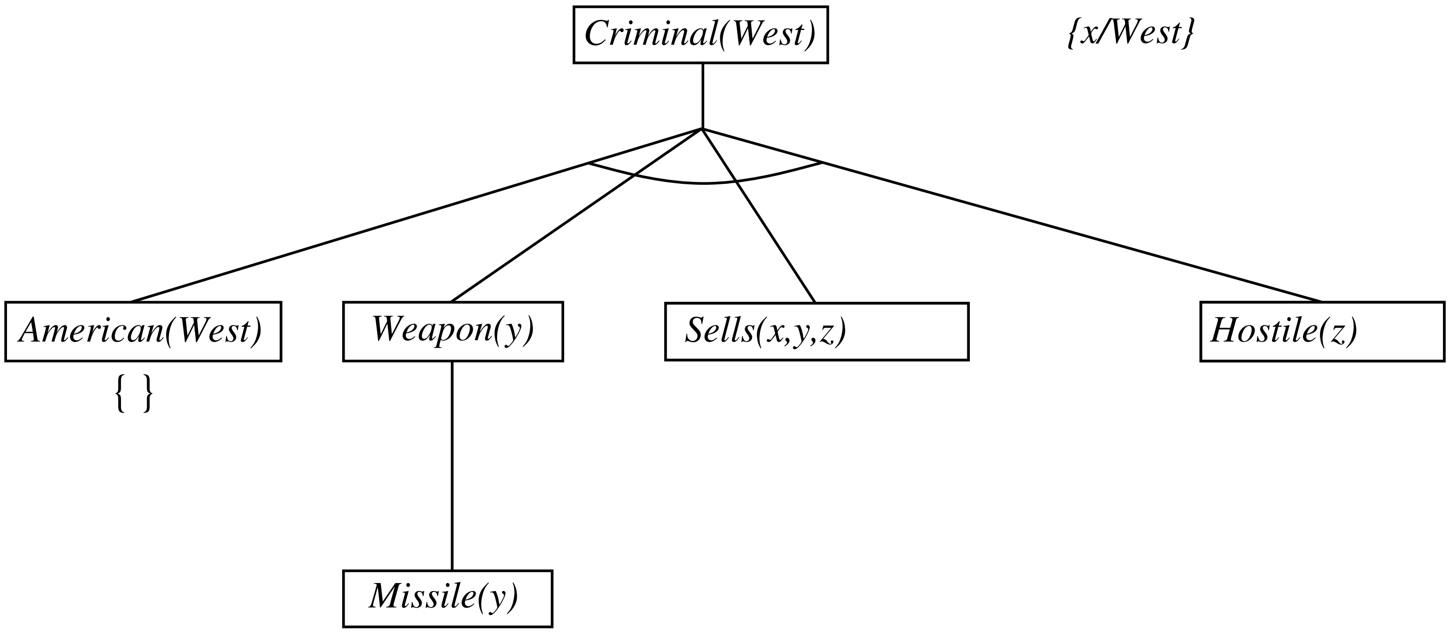
Example: backward chaining



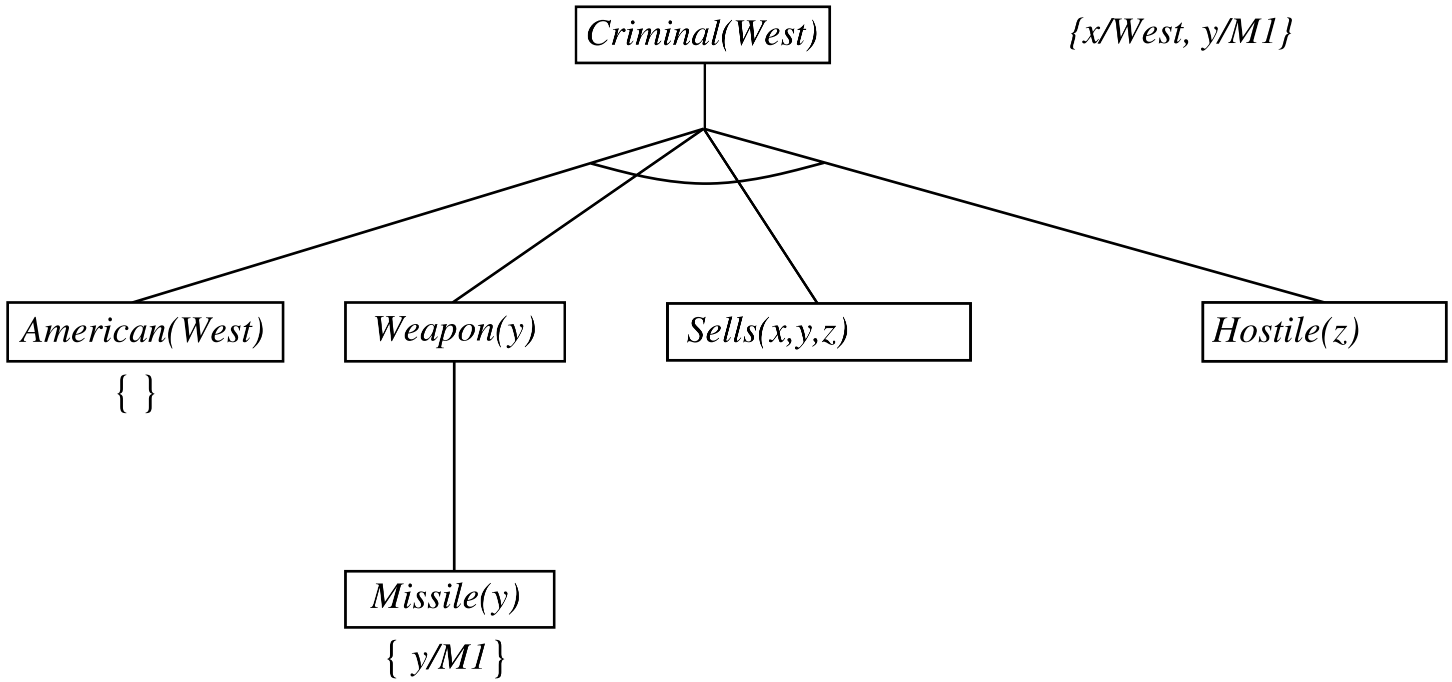
Example: backward chaining



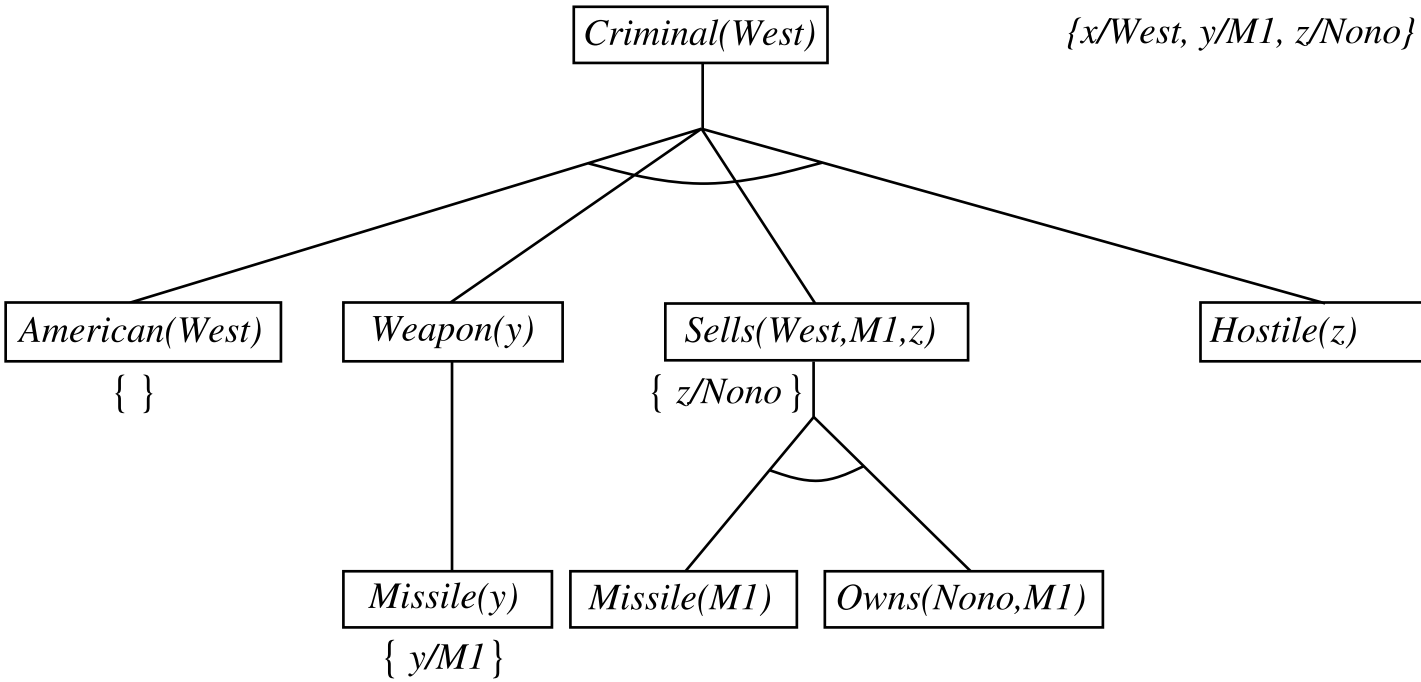
Example: backward chaining



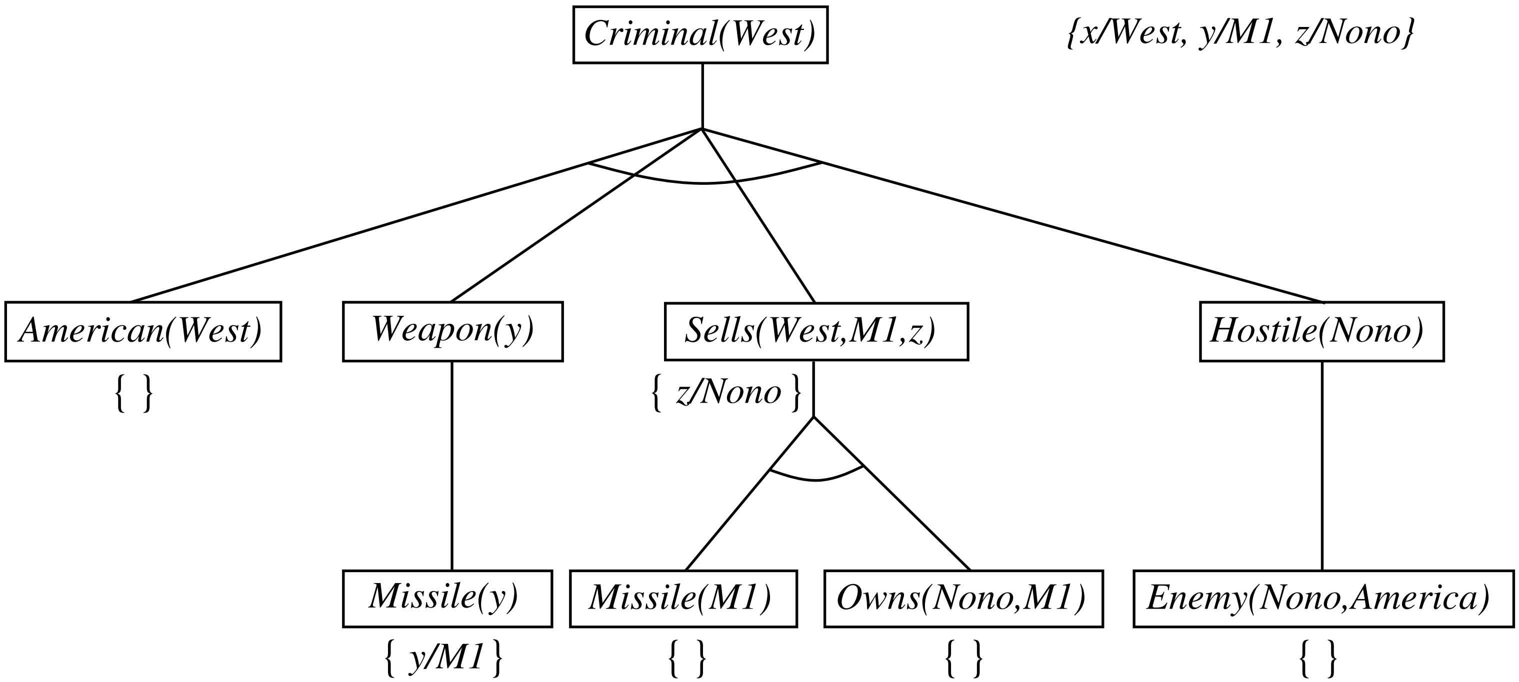
Example: backward chaining



Example: backward chaining



Example: backward chaining



Properties of backward chaining

Depth-first recursive proof search: space is linear in the size of proof

Incomplete due to infinite loops
$\Rightarrow$ fix by checking the current goal against every goal on the stack

Inefficient due to repeated subgoals (both success and failure)
$\Rightarrow$ fix using caching of previous results (extra space!)

Widely used for logic programming

First-order resolution

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{(\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)\theta}$$

where $\textsc{Unify}(\ell_i, \neg m_j) = \theta$.

E.g.

$$\frac{\begin{array}{l}\neg Rich(x) \vee Unhappy(x) \\ Rich(lin)\end{array}}{Unhappy(lin)}$$

with $\theta = \{x/lin\}$

Apply resolution steps to $CNF(KB \wedge \neg\alpha)$; complete for FOL

Conjunctive Normal Form

Any FOL KB can be converted to CNF

1. Replace $P \Rightarrow Q$ by $\neg P \vee Q$
2. Move $\neg$ inwards, e.g., $\neg \forall x\, P$ becomes $\exists x\, \neg P$
3. Standardize variables apart, e.g., $\forall x\, P \vee \exists x\, Q$ becomes $\forall x\, P \vee \exists y\, Q$
4. Move quantifiers left in order, e.g., $\forall x\, P \vee \exists x\, Q$ becomes $\forall x \exists y\, P \vee Q$
5. Eliminate $\exists$ by Skolemization (next slide)
6. Drop universal quantifiers
7. Distribute $\wedge$ over $\vee$, e.g., $(P \wedge Q) \vee R$ becomes $(P \vee Q) \wedge (P \vee R)$

Skolemization

$\exists x\, Rich(x)$ becomes $Rich(c)$ where c is a new Skolem constant

More tricky when $\exists$ is inside $\forall$
E.g., "Everyone has a heart"

$$\forall x \ \ .Person(x) \Rightarrow \exists y \ \ .Heart(y) \land Has(x, y)$$

Incorrect:

$$\forall x \ \ .Person(x) \Rightarrow Heart(H1) \land Has(x, H1)$$

Correct:

$$\forall x \ \ .Person(x) \Rightarrow Heart(H(x)) \land Has(x, H(x))$$

where H is a new symbol (Skolem function)

Skolem function arguments: all enclosing universally quantified variables

Conversion to CNF

Everyone who loves all animals is loved by someone:

$$\forall\, x \;\; .[\forall\, y \;\; Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists\, y \;\; Loves(y,x)]$$

1. Eliminate biconditionals and implications

$$\forall\, x \;\; .[\neg\forall\, y \;\; \neg Animal(y) \lor Loves(x,y)] \lor [\exists\, y \;\; Loves(y,x)]$$

2. Move $\neg$ inwards: $\neg\forall\, x, p \;\; \equiv \exists\, x \;\; \neg p$, $\;\; \neg\exists\, x, p \;\; \equiv \forall\, x \;\; \neg p$

$$\forall\, x \;\; .[\exists\, y \;\; \neg(\neg Animal(y) \lor Loves(x,y))] \lor [\exists\, y \;\; Loves(y,x)]$$
$$\forall\, x \;\; .[\exists\, y \;\; \neg\neg Animal(y) \land \neg Loves(x,y)] \lor [\exists\, y \;\; Loves(y,x)]$$
$$\forall\, x \;\; .[\exists\, y \;\; Animal(y) \land \neg Loves(x,y)] \lor [\exists\, y \;\; Loves(y,x)]$$

Conversion to CNF

3. Standardize variables: each quantifier should use a different one

$$\forall x\ .[\exists y\ \ Animal(y) \wedge \neg Loves(x, y)] \vee [\exists z\ \ Loves(z, x)]$$

4. Skolemize: a more general form of existential instantiation
   Each existential variable is replaced by a Skolem function
   of the enclosing universally quantified variables

$$\forall x\ .[Animal(F(x)) \wedge \neg Loves(x, f(x))] \vee Loves(g(x), x)$$

5. Drop universal quantifiers

$$[Animal(f(x)) \wedge \neg Loves(x, f(x))] \vee Loves(g(x), x)$$

6. Distribute $\wedge$ over $\vee$

$$[Animal(f(x)) \vee Loves(g(x), x)] \wedge [\neg Loves(x, f(x)) \vee Loves(g(x), x)]$$

Resolution derivation

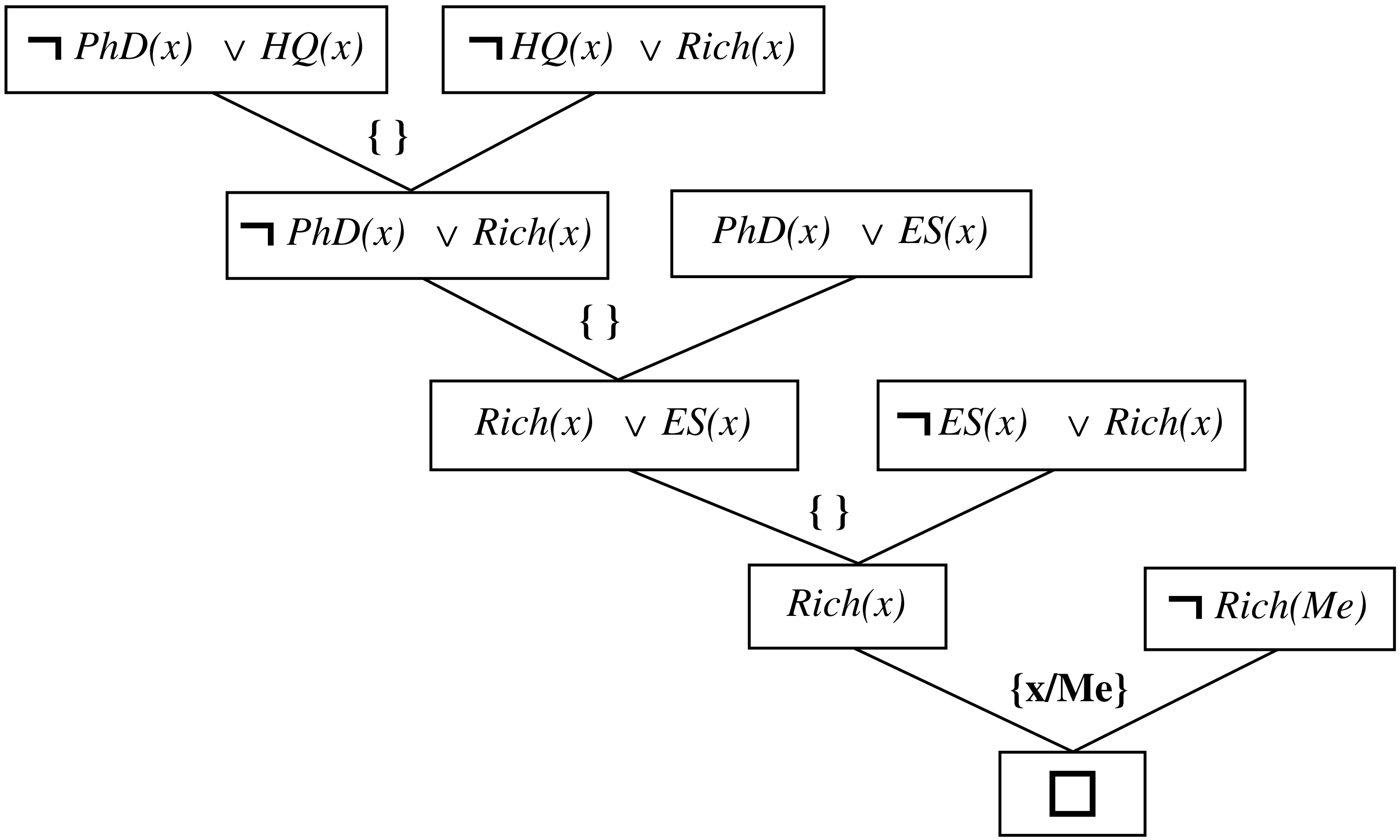
To prove α

– negate it
– convert to CNF
– add to CNF KB
– infer contradiction

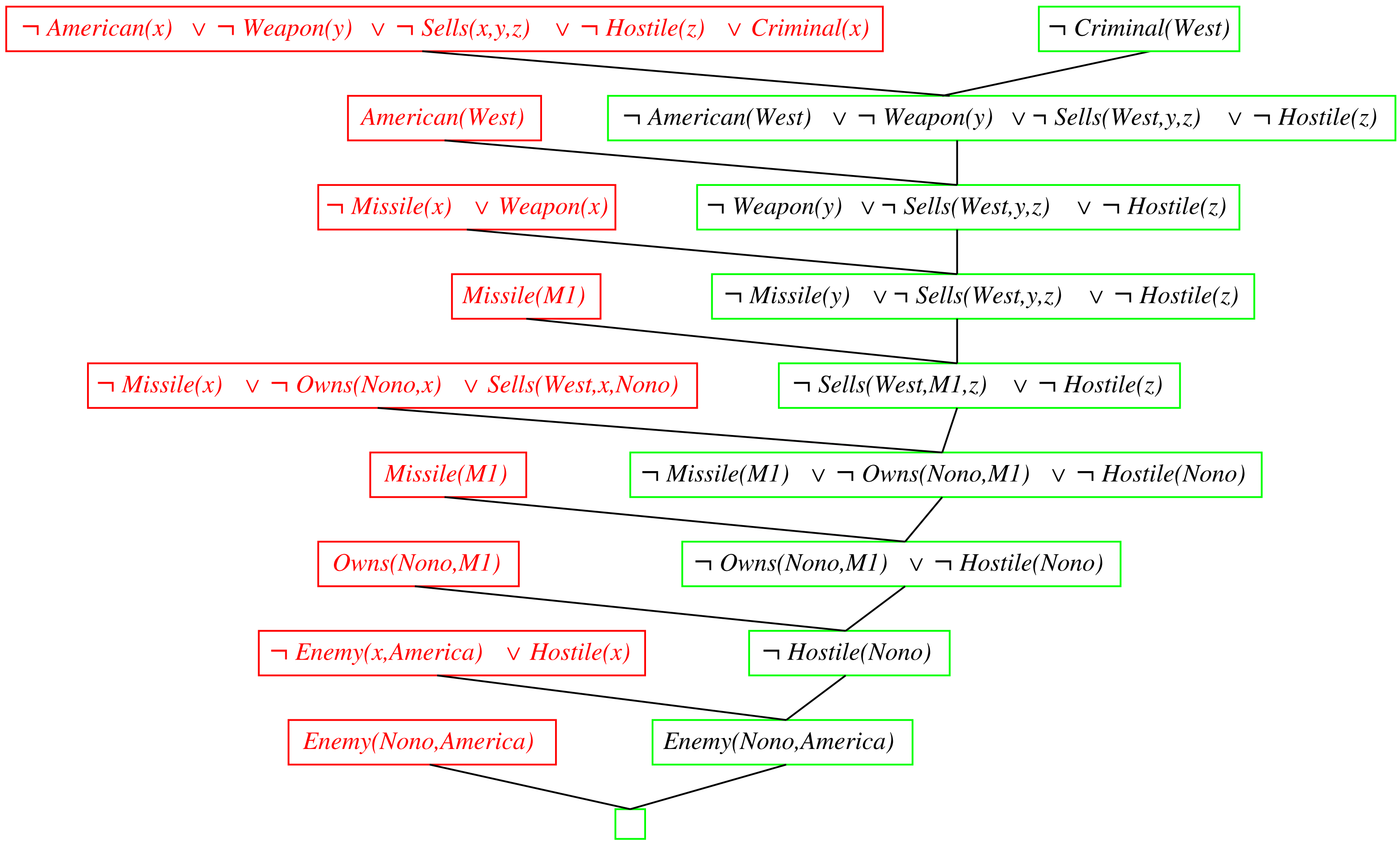
E.g., to prove $Rich(me)$, add $\neg Rich(me)$ to the CNF KB

$$\neg PhD(x) \vee HighlyQualified(x)$$
$$PhD(x) \vee EarlyEarnings(x)$$
$$\neg HighlyQualified(x) \vee Rich(x)$$
$$\neg EarlyEarnings(x) \vee Rich(x)$$

Example: resolution derivation



Resolution derivation: definite clauses



Completeness of resolution*

(Refutation) Completeness of resolution: If S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction

Proof sketch

– If S is unsatisfiable, then there exists a particular set of ground instances of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem)

– The **ground resolution theorem** is hold since propositional resolution is complete for ground sentences

– For any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained (lifting lemma)

Answer predicates*

In full FOL, we have the possibility of deriving $\exists x P(x)$ without being able to derive $P(t)$ for any t

Solution: answer-extraction process
  – replace query $\exists x P(x)$ by $\exists x(P(x) \wedge \neg A(x))$
  where A is a new predicate symbol, called the answer predicate
  – instead of deriving $\{\}$, derive any clause containing just the answer predicate
  – can always convert to and from a derivation of $\{\}$

E.g.,
$KB = \{Student(john), Student(jane), Happy(john)\}$
$Q = \exists x(Student(x) \wedge Happy(x))$
  $A(john)$, i.e., an answer is $john$

Hardness of resolution*

First-order resolution is not guaranteed to terminate

Propositional resolution is (determining if a set of clauses is satisfiable) NP-complete (Cook Theorem)

There are unsatisfiable clauses $\{c_1, c_2, \cdots, c_n\}$ s.t. the shortest derivation of $\{\ \}$ contains on the order of 2^n clauses (Haken, 1985)

Implications

– full theorem-proving may be too difficult

– need to consider other options

– – giving control to user, e.g., procedural representations

– – less expressive languages

e.g., Horn clauses (such as Prolog), semantic Web, knowledge graph

Resolution strategies*

strategies: reduce redundancy
– e.g., mathematical theorem proving, where we care about specific formulas
– automated theorem proving (ATP)
study strategies for automatically proving difficult theorems

- Unit preference
- Set of support
- Input resolution
- Subsumption
- Linear resolution, etc.

Ref. Chang C&Lee R, *Symbolic Logic and Mechanical Theorem Proving*, 2e, 1997

Model checking^{+}

Two efficient algorithms for propositional theorem proving based on model checking

Backtracking

– DPLL (Davis-Putnam-Logemann-Loveland) algorithm: recursive, depth-first enumeration of possible models

Local search

– Similarly, MIN-CONFLICTS for CSPs, using an evaluation function that counts the number of unsatisfied clauses

DPLL

DPLL: a complete backtracking algorithm
– improving TT-Entail

- Early termination: a clause is true if *any* literal is true
  E.g., $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless B, C
- Pure symbol heuristic: a pure symbol appears with the same "sign" in all clauses
  E.g., $(A \vee \neg B)$, $(\neg B \vee \neg C)$, $(C \vee A)$
  A (only positive appears) and B are pure, C is impure
  A sentence has a model $\rightarrow$ it has a model with the pure symbols assigned so as to make their literals true
- Unit clause heuristic: a unit clause with just one literal, with esp. clauses in which all literals but one are already assigned *false*
  E.g., if $B = true$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$
  assigning one unit clause can create another one (unit propagation)

DPLL algorithm#

```
def DPLL-SATISFIABLE?(s)
    inputs: s, a sentence in propositional logic

    clauses ← the set of clauses in the CNF representation of s
    symbols ← a list of the proposition symbols in s
    return DPLL(clauses, symbols, [ ])
```

```
def DPLL(clauses, symbols, model)

    if every clause in clauses is true in model then return true
    if some clause in clauses is false in model then return false
    P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
    if P is non-null then return DPLL(clauses, symbols − P, model ∪ {P = value})
    P, value ← FIND-UNIT-CLAUSE(clauses, model)
    if P is non-null then return DPLL(clauses, symbols − P, model ∪ {P = value})
    P ← FIRST(symbols); rest ← REST(symbols)
    return DPLL(clauses, rest, model ∪ {P = value}) or
              DPLL(clauses, rest, model ∪ {P = value})
```

Logic programming*

Computation as inference on logical KBs

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

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

Prolog*

Basis: backward chaining with Horn clauses + bells & whistles
Widely used in Europe, Japan (basis of 5th Generation prlinect)
Compilation techniques ⇒ approaching a billion LIPS

Program = set of clauses = `head :- literal`$_1$`, ... literal`$_n$`.`

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

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., given `alive(X) :- not dead(X).`
`alive(joe)` succeeds if `dead(joe)` fails

Example: Prolog program*

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]   B=[2]
         A=[1,2] B=[]
```

Answer set programming (ASP)*

Rule

$a \leftarrow b_1, \ldots, b_m$, not $c_1, \ldots$, not c_n

– a (head), b_i and c_j (body) are atoms

– true, if all literals to the body are true: a non-negated literal b_i is true if it has a derivation, a negated one, not c_j, is true if the atom c_j does not have one

Programs: finite collections of rules

ASP vs. Prolog*

Prolog: programming language

Need to understand Prolog's evaluation strategy, SLD resolution with unification

– the order of rules in a Prolog program and of subgoals (literals) in rule bodies matters

– Prolog misses true declarativity

ASP: specifications (yet do not allow the programmer to control the search)

– more declarative: it is intuitive, requires less background in logic, and its semantics is robust to changes in the order of literals in rules and rules in programs

– the ground program is fixed and only the data component changes

Automated theorem provers

Stanford Resolution Prover/FOL: one of the most mature subfields of ATP

E-prover (E 2.3, github.com/eprover): one of the SOTA FOL /w equality prover

TPTP (Thousands of Problems for Theorem Provers) problem library

CADE ATP System Competition (CASC): a yearly competition of first-order systems

Proof assistant (interactive theorem prover): a software tool to assist with the development of formal proofs by human-machine collaboration

— LEAN, Coq, HOL, Isabelle, etc.

LEAN*

Input: a formal language for expressing math statements (definitions, axioms, conjectures, theorems, and constructions) in a human-readable and machine-verifiable format

Proof assistant: LEAN serves as a proof assistant, allowing users to interactively develop and verify math proofs (correctness and consistency)

Automated reasoning: the resolution-based automated reasoning engine is used to automate the process of proof

Proof checking: The resolution-based proofs are checked for correctness and consistency

Output: Upon successful verification, LEAN provides formalized math theorems and constructions, along with their proofs, in a machine-verifiable format