Overview

Inference in First-Order Logic

- Propositional vs. First-Order Inference
- Unification and First-Order Inference
- Forward Chaining
- Backward Chaining
- Resolution

Propositional vs. First-Order Inference

- Since propositional inference is very efficient (e.g. WalkSAT), an inference idea is to convert first-order logic into propositional logic and use propositional inference
  - Eliminate universal quantifiers
  - Eliminate existential quantifiers
  - Replace praedicate symbols by constants
  - Replace function symbols
- Results:
  - Approach works
  - However, since there are infinite ground symbols for a function, inference is only semi-decidable
    - (e.g. father-of (father-of (father-of (… John))))

Universal Instantiation

- An universal quantified variable can be substituted with every ground term
- Let Subst (θ,α) denote the result of applying the substitution θ to the sentence α
- **Rule**: Every instantiation of a universal quantified sentences follow from the sentence:
  - For every variable v and every ground term g: $\frac{\forall v\ \alpha}{\text{SUBST}(\{v/g\},\alpha)}$

- *Example:* $\forall$ x King (x) $\wedge$ Greedy (x) $\Rightarrow$ Evil (x) yields:
  - King (John) $\wedge$ Greedy (John) $\Rightarrow$ Evil (John)
  - King (Richard) $\wedge$ Greedy (Richard) $\Rightarrow$ Evil (Richard)
  - King (Father(John)) $\wedge$ Greedy (Father(John)) $\Rightarrow$ Evil (Father(John))
  - …

Existantial Instantiation

- An existantial quantified variable can be substituted with a single new constant symbol
- **Rule**: For any sentence α, variable v and constant symbol k, that does not appear elsewhere in the knowledge base:

$$\frac{\exists v \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

- The new symbol is called **Skolem constant**

- *Example:* ∃ x Crown (x) ∧ OnHead (x, John) yields:

  Crown (C_1) ∧ OnHead (C_1, John)

- Exception: If the existantial quantified variable depends on a universal quantified variable, a **Skolem function** instead of a Skolem constant is needed:
- *Example:* ∀ x Monarchy (x) ∃ y Crown (y) ∧ OnHead (y, KingOf (x)) yields:
  - Monarchy (England) ∧ Crown (CrownOf (England)) ∧ OnHead (CrownOf (England), KingOf (England))
  - Monarchy (Thailand) ∧ Crown (CrownOf (Thailand)) ∧ OnHead (CrownOf (Thailand), KingOf (Thailand))
  - ….

Universal and Existential Instantiation

- Universal instantiation can be applied many times to the same axiom to produce different consequences.
- Existential instantiation can be applied just once and replaces the original sentence.
  - Example: We can replace $\exists x$ *Kill (x, Victim)* by *Kill (Murderer, Victim)*
  - The new knowledge base is not equivalent to the old one, but it is satisfiable, if the old one is satisfiable.

Predicates and Functions

- Predicates can be easily replaced by constants
  - Example: King (John) $\wedge$ Greedy (John) $\Rightarrow$ Evil (John) is replaced by
    JohnIsKing $\wedge$ JohnIsGreedy $\Rightarrow$ JohnIsEvil
- Functions can also be easily replaced by constants
  - Example Father(John) is replaced by FatherJohn
  - Problem: From a function, infinite many ground terms can be generated:
    - FatherJohn, FatherFatherJohn, FatherFatherFatherJohn, etc.
    - It is impossible to generate all of them
  - Solution: Generate a subset of the infinite many ground terms for inference. If it fails, generate a larger subset etc. (see next slide)

Propositional Inference

- Conjecture: Every First-Order Logic (FOL) Knowledge Base (KB) can be converted into a propositional KB, so that entailment is preserved.
  - Conversion procedure as described above
  - Apply resolution to the converted WB to infer result
  - Problem: Infinite many function symbols
- Theorem (Herbrand 1930): If a sentence α follows from a FOL-KB, than it follows also from a finite subset of the propositional KB
  - Proof idea:
    - generate from n = 0 to infinite propositional WBs by instantiating function symbols to the depth of n and proof α in them.
    - If α follows, o.k.; if not indefinite loop
  - Conclusion: **First-Order Logic is semi-decidable**!

First-Order Inference: Generalized Modus Ponens

- Propositionalization generate many irrelevant sentences
  - e.g. from the sentence $\forall$ x King (x) $\wedge$ Greedy (x) $\Rightarrow$ Evil (x) and 100 entities (John, Richard, Crown, England etc.) 100 instances are generated, although most of them are useless for the intended inference.
- Better idea: Make only those instantiations, that are necessary for the rule to fire: **Generalized Modus Ponens**
  - For atomic sentences p_i, p_i' and q, where there is a substitution θ such that

Subst (θ, p_i') = Subst (θ, p_i), for all i:

$$\frac{p_1', \; p_2', \; \ldots, \; p_n', \; (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

  - Example:
    - $\forall$ x King (x) $\wedge$ Greedy (x) $\Rightarrow$ Evil (x)
    - $\forall$ y Greedy (y),
    - King (John)

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 $Evil(x)$
SUBST(θ, q) is $Evil(John)$.

Unification

- Process of finding substitutions, that make different logical expressions look identical.
- Unify (p, q) = θ where Subst (θ, p) = Subst (θ, q)
- Examples:
  - Unify (Knows (John, x) (Knows (John, Jane)) = {x/Jane}
  - Unify (Knows (John, x) (Knows (y, Bill)) = {x/Bill, y/John}
  - Unify (Knows (John, x) (Knows (y, Mother(y))) = {y/John, x/Mother(John)}
  - Unify (Knows (John, x) (Knows (x, Elizabeth)) = failure
    - in last unification, variable renaming is necessary, e.g. (Knows (y, Elizabeth)) for success
- General solution for naming conflicts: **Standardizing apart** (naming differently) of all variables

Unification Algorithm


```
function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS(x), ARGS(y), UNIFY(OP(x), OP(y), θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y), θ))
  else return failure

function UNIFY-VAR(var, x, θ) returns a substitution
  if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

- Arguments x and y can be any expression
- Argument θ is the resulting substitution, initially empty
- Occur-Check? (var, x)
  - is var part of x?
  - e.g. var = y & x = f(y)
  - increases complexity of unification from O(n) to $O(n^2)$
  - Omitted in logical programming languages like PROLOG

Example for Unification Algorithm

- Knows (John, x) and Knows (John, Jane)

//both expressions are compound

➢return (Unify (Args [x], Args [y], Unify (Op [x], Op [y], θ)))

[x] = Knows (John, x) and [y] = Knows (John, Jane) with θ = {}

Unify (Op [x], Op [y], θ) = Unify (Knows, Knows) {} = {}

Unify (Args [x], Args [y], θ) = Unify ((John, x), (John, Jane) {})

//both expressions are lists

➢return (Unify (Rest [x], Rest [y], Unify (First [x], First [y], θ)))

[x] = (John, x) and [y] = (John, Jane)

Unify ((x), (Jane), Unify (John, John, {})) = Unify ((x), (Jane), {}) = Unify (x, Jane, {})

//x is variable

➢return (unify-var (x, Jane, {}) = **{x/Jane}**

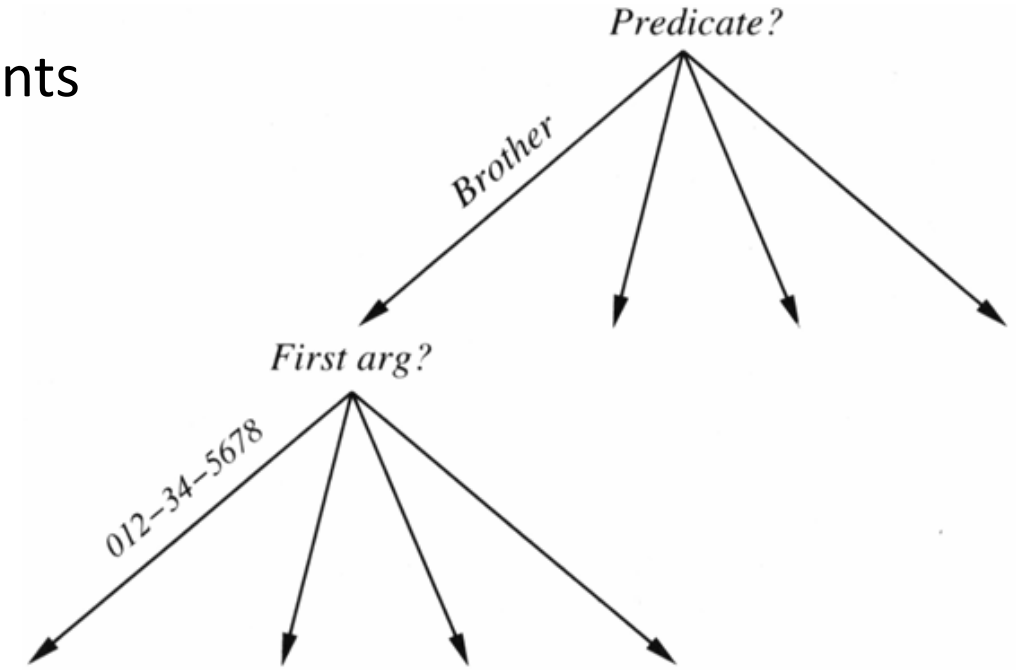
Fast Storage and Retrieval with Indices

- In a large knowledge base, there are many different predicates, and even one predicate may have many different arguments
  - Efficiency can be increased, if the unification algorithm compares only those predicates having a chance of matching
    - Knows (John, x) and Employs (IBM; Richard) have different predicates and it makes no sense to check, whether they can match
    - Predicate indexing puts all facts with the same precidate in one bucket, e.g.

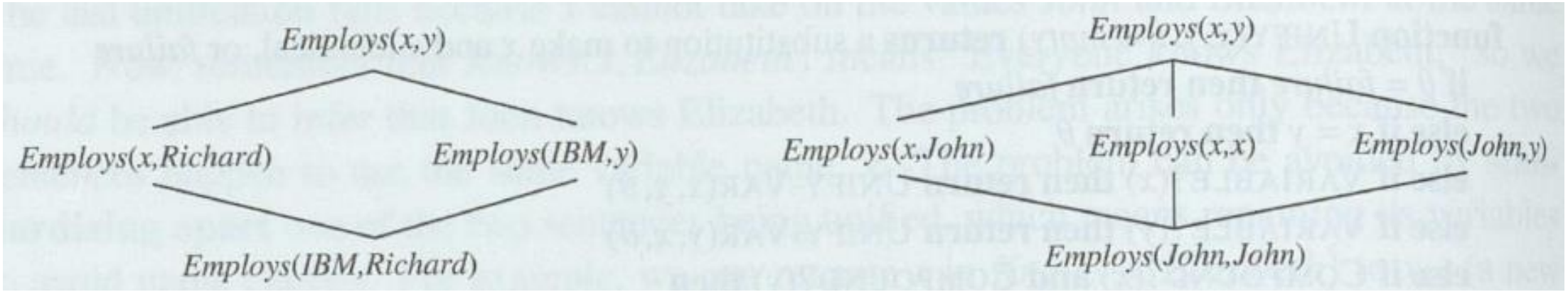
Key	Positive	Negative	Conclusion	Premise
Brother	$Brother(Richard, John)$ $Brother(Ted, Jack)$ $Brother(Jack, Bobbie)$	$\neg Brother(Ann, Sam)$	$Brother(x,y) \wedge Male(y) \Rightarrow Brother(y,x)$	$Brother(x,y) \wedge Male(y) \Rightarrow Brother(y,x)$ $Brother(x,y) \Rightarrow Male(x)$
Male	$Male(Jack)$ $Male(Ted)$ $\ldots$	$\neg Male(Ann)$ $\ldots$	$Brother(x,y) \Rightarrow Male(x)$	$Brother(x,y) \wedge Male(y) \Rightarrow Brother(y,x)$

Fast Storage and Retrieval with Indices

- For frequent used predicates, an index on the arguments further increases efficiency:



- More elaborate argument indexing based on a subsumption lattice containing indices for all possible queries that unify with them, e.g:



Forward and Backward Chaining

- Similar to propositional logic, rules (definite clauses, like King (x) $\wedge$ Greedy (x) $\Rightarrow$ Evil (x)) can be used for forward and backward chaining with the generalized Modus Ponens.
  - Existential Quantifier are not allowed (must be replaced by Skolom constants or functions)
  - Universal quantifiers are implicit (every variable is universally quantified)
- Generalization of propositional algorithms for First Order Logic
  - **Forward Chaining:** Infer everything and check, whether the goal (query) is included
  - **Backward Chaining:** Infer necessary sentences backward from the goal (query)

Example

- The lay says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.
- Prove, that Colonel West is a criminal.

- First the text is transformed in rules (first-order definite clauses):
  - American (x) $\wedge$ Weapon (y) $\wedge$ Sells (x,y,z) $\wedge$ Hostile (z) $\Rightarrow$ Criminal (x)
  - $\exists$ x Owns (Nono x) $\wedge$ Missile (x)
    - Owns (Nono, M1)
    - Missile (M1)
  - Missile (x) $\wedge$ Owns (Nono, x) $\Rightarrow$ Sells (West, x, Nono)
  - Missile (x) $\Rightarrow$ Weapon (x) // background knowledge
  - Enemy (x, America) $\Rightarrow$ Hostile (x)
  - American (West)
  - Enemy (Nono, America)

Simple Forward Chaining Algorithm

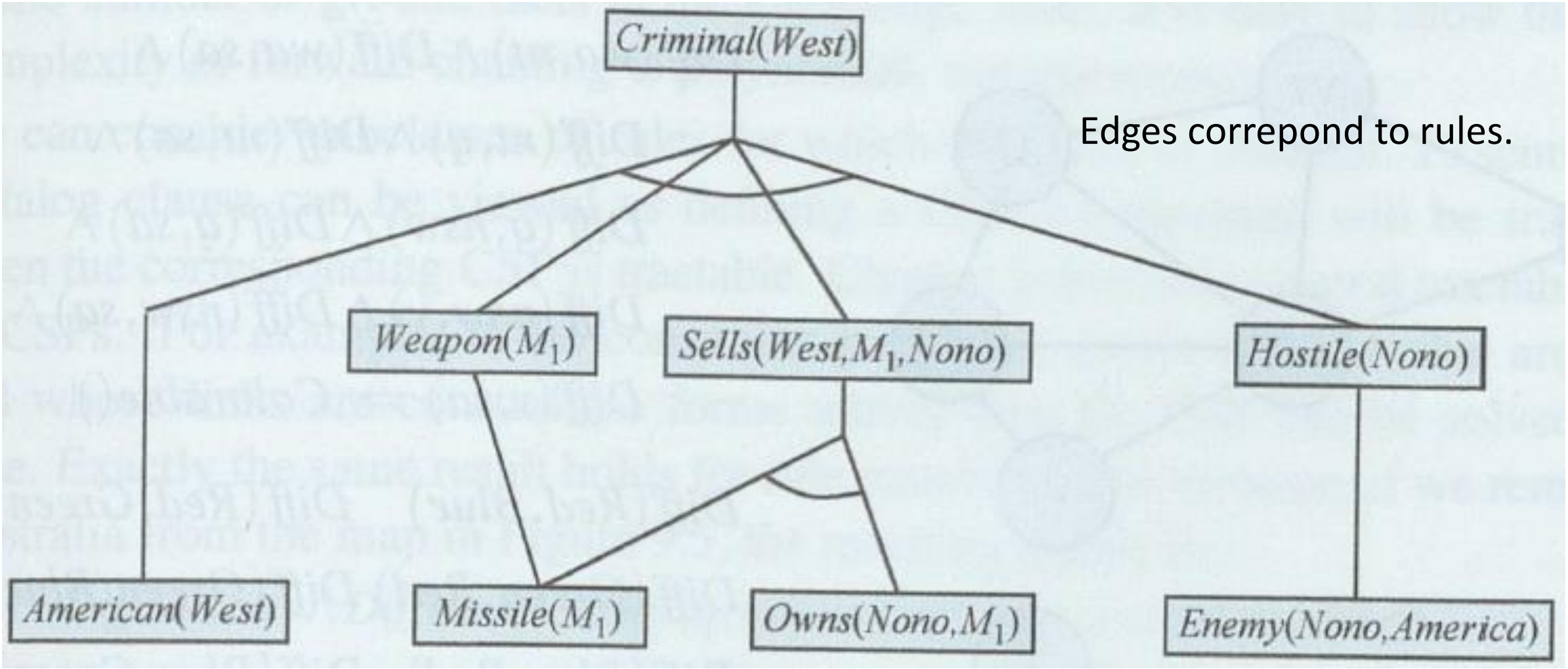
function FOL-FC-ASK(KB, α) **returns** a substitution or *false*
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, the query, an atomic sentence

  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 such that 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*
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if *new* $= \{\}$ **then return** *false*
    add *new* to KB

//Variable renaming

- On each iteration, it adds to KB all atomic sentences that can be inferred in one step.
- straightforward, but inefficient

Proof Tree Generated by Forward Chaining



Edges correpond to rules.

Initial facts at bottom level, facts inferred on the first (second) iteration in middle (top) level.

Efficient Forward Chaining

- Three sources of inefficiency in the simple Forward Chaining Algorithm:
  - Per iteration (inner loop) the algorithm tries to mach every rule against every fact in KB
  - The algorithm rechecks every rule on every iteration, even if few additiong have been made
  - The algorithm generates many facts that are irrelevant to the goal

Matching Rules against Known Facts

- Consider a rule like
  - Missile (x) $\wedge$ Owns (Nono, x) $\Rightarrow$ Sells (West, x, Nono)
- The interpreter checks for the first premise (Missile (x)) and finds all matches in nearly constant time due to indexing.
- For each match, it has to check the second premise (Owns (Nono, x)).
  - If there are many matches for the first premise, many matches for the second premise are necessary.
  - If the second premise has only few matches, it would be more efficient, to start with the second premise.
  - Improvement: **Conjunct ordering** with heuristic (e.g. according size in index data base)

Incremental Forward Chaining

- In each iteration (except the first one), it is beneficial, to check only those rules, which have premises containg facts, that of changed in the last iteration.
  - Similar idea in PL-FC-Entails for propositional rules: Each newly inferred fact send messages to all „its“ rules and each rule had a count for unfulfilled premises; if the count went 0, the rule fired
  - More complicated for rules with variables: partial rule matches can be stored
    - Rete Algorithm with data flow network, where each node is a literal from a rule premise. Variable binding flow through the network are may be filtered
    - Example (circles: predicate tests; squares: filters; rectangles: actions)

Rules:

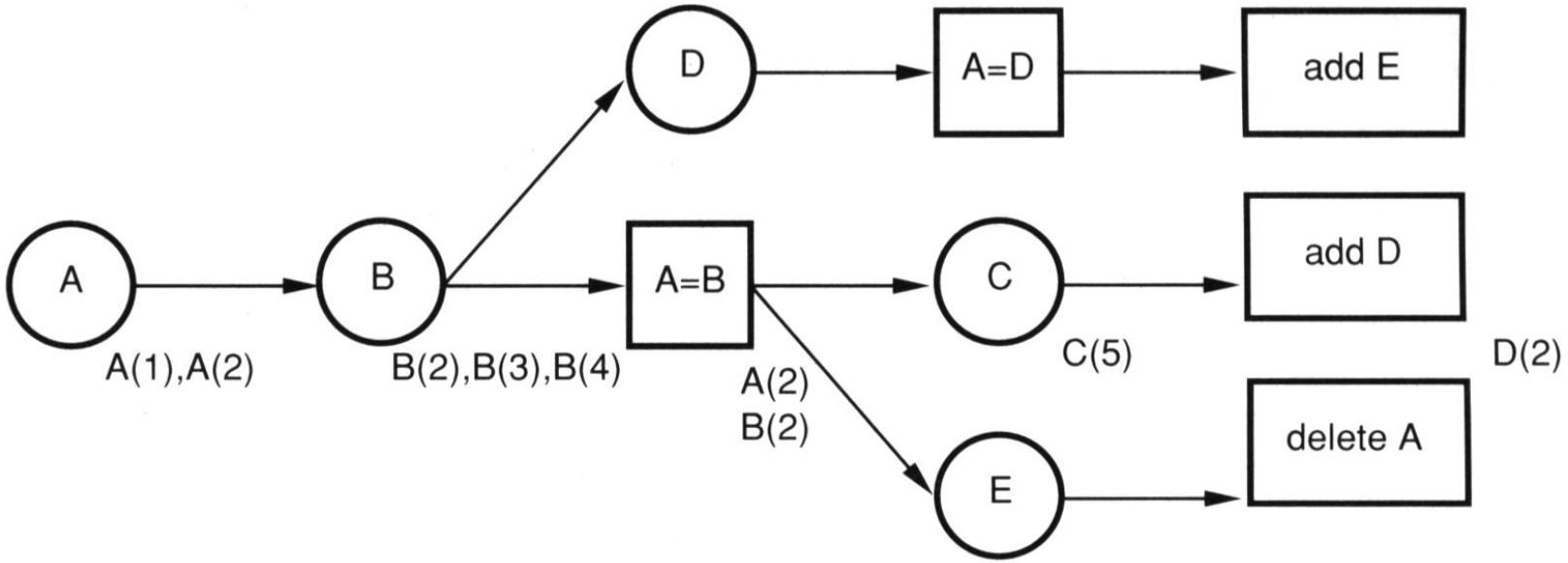
$A(x) \wedge B(x) \wedge C(y) \Rightarrow$ add $D(x)$

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

$A(x) \wedge B(x) \wedge E(x) \Rightarrow$ delete $A(x)$

Working Memory:

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



Irrelevant Facts

- The most elegant way to avoid deriving irrelevant facts is backward chaining.
- Another way is the **magic set approach** originating from the field of deductive databases using forwand chaining as the standard inference tool (instead of SQL queries)
  - Idea: Rewrite a rule using information from the goal, so that only relevant variable bindings – those belonging to the magic set – are considerd during forward inference
  - Example: If the the goal is to prove Criminal (West), the rule that concludes Criminal (x) will be rewritten to include an extra conjunct:
    - $\text{Magic}(x) \wedge \text{American}(x) \wedge \text{Weapon}(x) \wedge \text{Sells}(x,y,z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$
    - In addition, the fact „Magic (West)“ is added to the knowledge base.
- Integration of elements of backward chaining in forward chaining by a preprocessing step.
- Implemented in **Datalog** (query language for deductive databases; a subset of PROLOG)

Backward Chaining

Like propositional backward chaining, FOL-backward chaining is a kind of AND/OR search:

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

function FOL-BC-OR(KB, goal, θ) returns a substitution
  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'

function FOL-BC-AND(KB, goals, θ) returns a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else
    first, rest ← FIRST(goals), REST(goals)
    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''
```

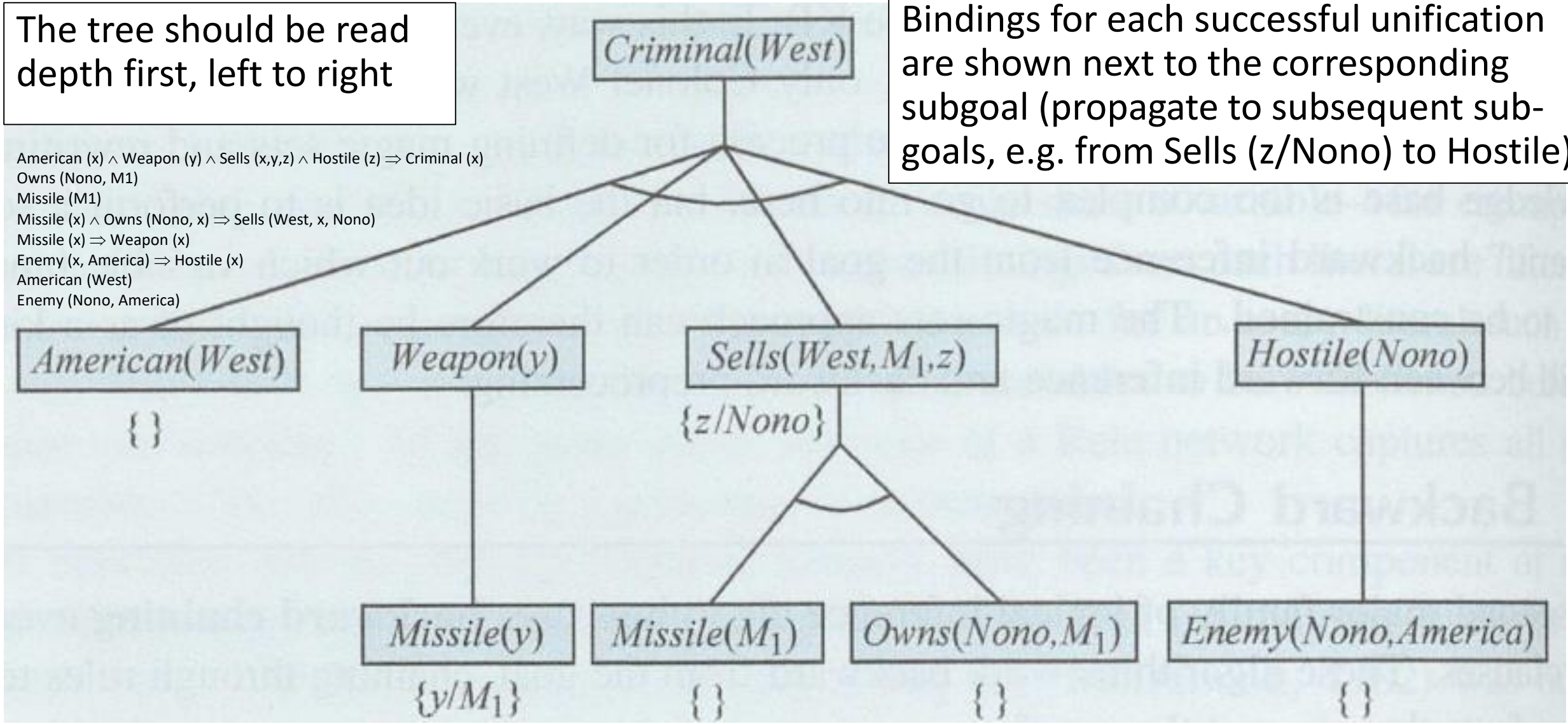
- OR-Search: A goal can be proved by any rule
- AND-Search: All conjuncts of a rule must be proved
- Unknown conjuncts cause a recursive call of the search
- Algorithm produces multiple solutions one after the other (or failure)

Proof Tree for Backward Reasoning

The tree should be read depth first, left to right

Bindings for each successful unification are shown next to the corresponding subgoal (propagate to subsequent sub-goals, e.g. from Sells (z/Nono) to Hostile)

American (x) ∧ Weapon (y) ∧ Sells (x,y,z) ∧ Hostile (z) ⇒ Criminal (x)
Owns (Nono, M1)
Missile (M1)
Missile (x) ∧ Owns (Nono, x) ⇒ Sells (West, x, Nono)
Missile (x) ⇒ Weapon (x)
Enemy (x, America) ⇒ Hostile (x)
American (West)
Enemy (Nono, America)



Properties of Backward Chaining

- Depth-first search algorithm:
  - Linear space requirements
  - Problems with repeated states and incompleteness (unlike forward chaining)
  - Might check subgoals multiple times
- Nevertheless popular and effective in logic programming languages (with some adaptions)

Logic Programming Languages

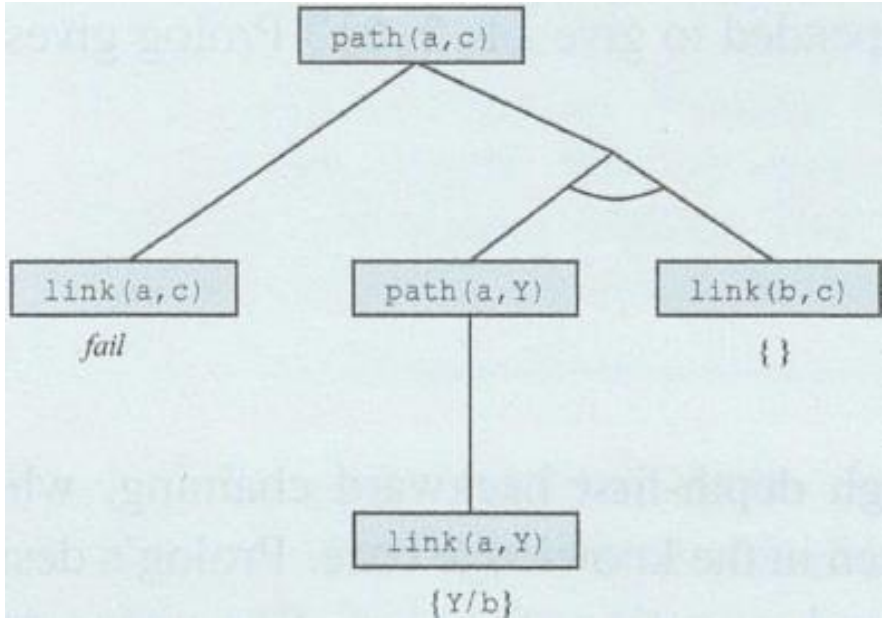
- **Algorithm = Logic + Control**
- Naive implementations of inference procedures are too slow, therefore logic programming languages offer mechanisms to control the inference process
- Most popular example: PROLOG
- Example definition:
  - member (X, [X | L]) // X is a member of a list, whose first element is X
  - member (X, [Y | L]) :- member (X, L)
    // X is member of a list, whose first element Y ≠ X, if X is a member of the rest of the list
  - Notation corresponds to first-order logic sentences:
    - $\forall$ x, l Member (x, [x | l])
    - $\forall$ x, y, l Member (x, l) $\Rightarrow$ Member (x, [y | l])
- Another example definition in PROLOG
  - append ([ ], Y, Y] // appending the empty list to Y yields Y
  - append ([A|X], Y, [A,Z] :- append [X, Y, Z]
    // [A|Z] is the result of appending [A|X] and Y, if Z is the result of appending X and Y

Properties of PROLOG

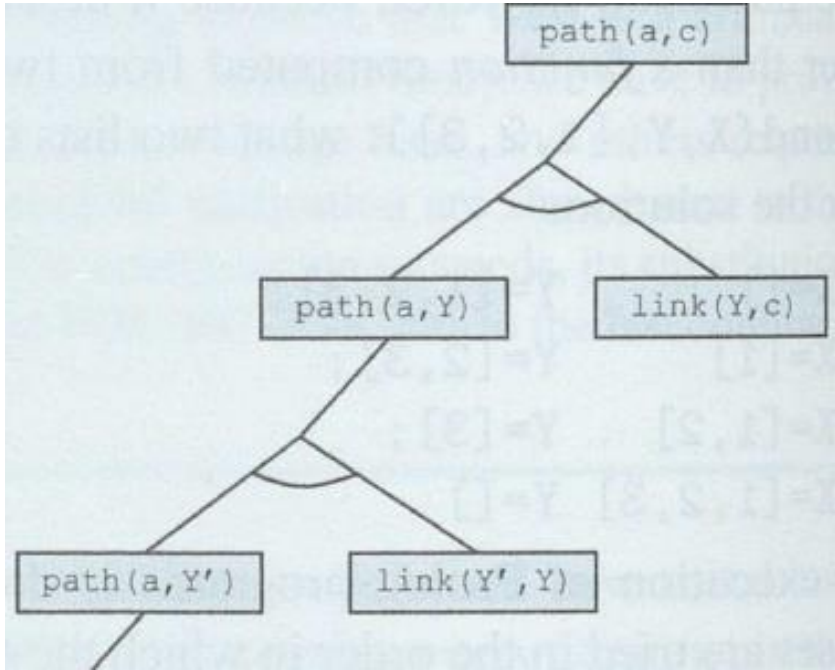
- PROLOG uses database semantics (unique name assumption, closed-world assumption, domain closure)
  - Negation as failure: The goal ¬P is viewed as proved, if P cannot be inferred
  - Syntactically disjunct terms are disjunct, i.e. it cannot be stated that A=B or A= f(x), if A is a constant
- Program consists of sequence of rules (definite clauses), all variables are universal quantified and disjunct in different sentences.
- There are built-in functions for arithmetic and built-in predicates with side effects when executed, e.g. for input/output and modifying the knowledge base (assert/retract predicates)
- Occur check is omitted in PROLOG‘s unification algorithm
- PROLOG uses depth-first backward chaining search with no checks for infinite recursion

Infinite Loops

- Prolog program for finding a path between two nodes:
  - path (X,Z) :- link (X,Z)
  - path (X,Z) :- path (X,Y), link (Y,Z)
  - link (a,b)
  - link (b,c)
- Anfrage: path (a,c)
- Antwort: yes (s. inference tree)

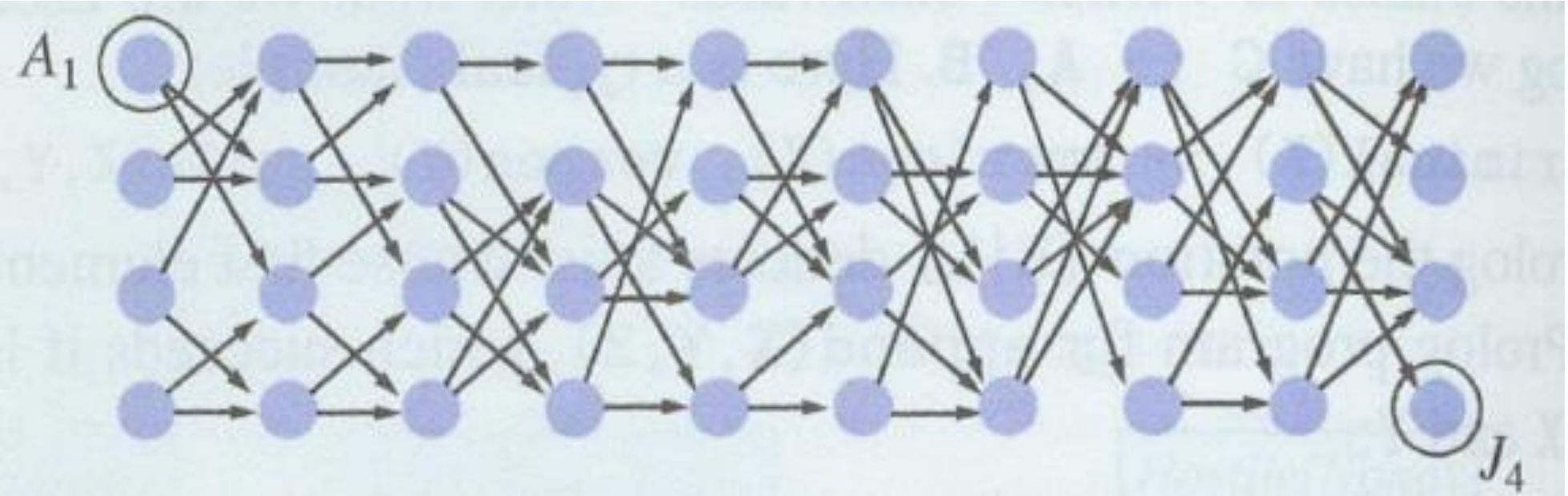


- But: Wrong order of both clauses causes an infinite loop:
  - path (X,Z) :- path (X,Y), link (Y,Z)
  - path (X,Z) :- link (X,Z)



Redundant Inference

- Problem: To find a path from A_1 to J_4 in a graph, where all elements of one column (A_1 to A_4) are connected with the next column, PROLOG needs 877 inference steps, while forward chaining would need 62 inferences only.
  - Forward chaining on graph search problems is an example of **dynamic programming**, in which a solution to subproblems are constructed incrementally from those of smaller subproblems and cached to avoid recomputation.
- Solution: **Tabled logic programming** with efficient storage and retrieval mechanisms combining goal-directedness of backward chaining with dynamic programming efficiency of forward chaining.



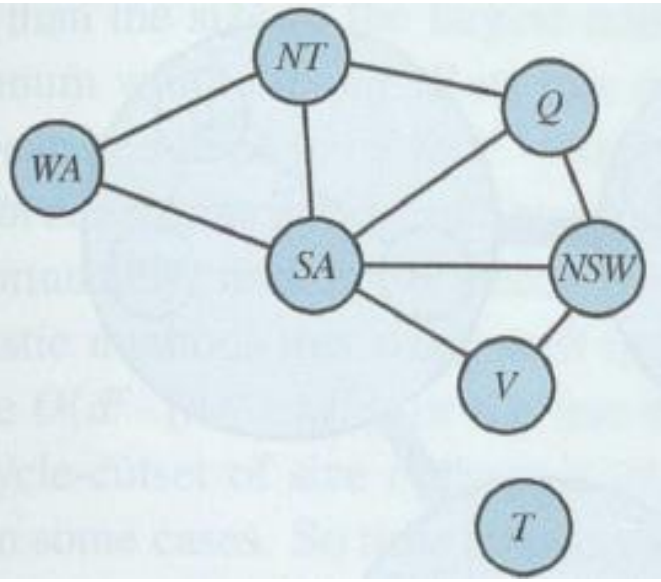
Logic Programming

Computation as inference in knowledge bases (compare „Knowledge engineering“ in chap. 8)

Logic Programming	**Standard Programming**
Identify Problem	Identify Problem
Gather Informations	Gather Informations
	Design Solution
Code Information in KB	Program Solution
Code problem instance as facts	Code problem instance as data
Ask queries	Apply program on data
Find erroneous facts / rules	Debugging

It should be easier to find an error in „Capital (Bonn, Germany)“ than in „x:= x+2“

Constraint Logic Programming



- Constraint satisfaction problems (CSPs) with **finite domains** can be encoded as definite clauses
  - *Example (map coloring problem): Diff (wa,nt) ∧ Diff (wa, sa) ∧ Diff (nt,q) ∧ Diff (nt,sa) ∧ Diff (q,nsw) ∧ Diff (q,sa) ∧ Diff (nsw,v) ∧ Diff (nsw,sa) ∧ Diff (v,sa) ⇒ Colorable ( )*
  - PROLOG can be used to solve them
- **Infinite domain CSPs** (e.g. with numeric values) require different algorithms
  - Example: *triangle (X,Y,Z) :- X>0, Y>0, Z>0, X+Y>Z, Y+Z>X, X+Z>Y*
  - *Triangle (3,4,5)* suceeds in PROLOG, but *Triangle (3,4,Z)* cannot be solved (Solution is: 7 > Z > 1)
- **CLP systems** incorperate various constraint-solving algorithms, e.g. linear programming algorithms, constraint satisfaction algorithms, logic programming, deductive databases.

Resolution

- Resolution as a complete inference system not only works for propositional logic, but can be extended to first-order logic (FOL) as well.
  1. Convert FOL-sentences in conjunctive normal form (CNF)
  2. Apply FOL resolution inference rule

Converting FOL-sentences in CNF

- Eliminate implications
- Move $\neg$ inwards
- Stanardize variables (and move quantifiers left)
- Skolemize (remove existential quantifiers)
- Drop universal quantifieres
- Distribute $\vee$ over $\wedge$ (and flatten nested conjunctions and disjunctions)

Converting FOL-sentences in CNF (1)

- **Eliminate implications**: Recall that $p \Rightarrow q$ is the same as $\neg p \vee q$. So replace all implications by the corresponding disjunctions.
- **Move $\neg$ inwards**: Negations are allowed only on atoms in conjunctive normal form, and not at all in implicative normal form. We eliminate negations with wide scope using de Morgan's laws (see Exercise 6.2), the quantifier equivalences and double negation:

$$\begin{aligned}
\neg(p \vee q) &\quad \text{becomes} \quad \neg p \wedge \neg q \\
\neg(p \wedge q) &\quad \text{becomes} \quad \neg p \vee \neg q \\
\neg \forall x, p &\quad \text{becomes} \quad \exists x \ \neg p \\
\neg \exists x, p &\quad \text{becomes} \quad \forall x \ \neg p \\
\neg\neg p &\quad \text{becomes} \quad p
\end{aligned}$$

- **Standardize variables**: For sentences like $(\forall x \ P(x)) \vee (\exists x \ Q(x))$ that use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers.
- **Move quantifiers left**: The sentence is now in a form in which all the quantifiers can be moved to the left, in the order in which they appear, without changing the meaning of the sentence. It is tedious to prove this properly; it involves equivalences such as

$$p \vee \forall x \ q \quad \text{becomes} \quad \forall x \ p \vee q$$

which is true because p here is guaranteed not to contain an x.

Converting FOL-sentences in CNF (2)

♢ **Skolemize**: **Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Elimination rule of Section 9.1—translate $\exists x \;\; P(x)$ into $P(A)$, where A is a constant that does not appear elsewhere in the KB. But there is the added complication that some of the existential quantifiers, even though moved left, may still be nested inside a universal quantifier. Consider "Everyone has a heart":

$$\forall x \;\; Person(x) \Rightarrow \exists y \;\; Heart(y) \wedge Has(x, y)$$

If we just replaced y with a constant, H, we would get

$$\forall x \;\; Person(x) \Rightarrow Heart(H) \wedge Has(x, H)$$

which says that everyone has the same heart H. We need to say that the heart they have is not necessarily shared, that is, it can be found by applying to each person a function that maps from person to heart:

$$\forall x \;\; Person(x) \Rightarrow Heart(F(x)) \wedge Has(x, F(x))$$

where F is a function name that does not appear elsewhere in the KB. F is called a **Skolem function**. In general, the existentially quantified variable is replaced by a term that consists of a Skolem function applied to all the variables universally quantified *outside* the existential quantifier in question. Skolemization eliminates all existentially quantified variables, so we are now free to drop the universal quantifiers, because any variable must be universally quantified.

Converting FOL-sentences in CNF (3)

- **Distribute $\wedge$ over $\vee$:** $(a \wedge b) \vee c$ becomes $(a \vee c) \wedge (b \vee c)$.
- **Flatten nested conjunctions and disjunctions:** $(a \vee b) \vee c$ becomes $(a \vee b \vee c)$, and $(a \wedge b) \wedge c$ becomes $(a \wedge b \wedge c)$.
  At this point, the sentence is in conjunctive normal form (CNF): it is a conjunction where every conjunct is a disjunction of literals. This form is sufficient for resolution, but it may be difficult for us humans to understand.
- **Convert disjunctions to implications:** Optionally, you can take one more step to convert to implicative normal form. For each conjunct, gather up the negative literals into one list, the positive literals into another, and build an implication from them:
  $(\neg a \vee \neg b \vee c \vee d)$ becomes $(a \wedge b \Rightarrow c \vee d)$

Example1 for CNF Conversion

$American(x) \wedge Weapon(y) \wedge Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x)$

$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x,y,z) \vee \neg Hostile(z) \vee Criminal(x).$

$Owns(Nono,M1)$ and $Missile(M1)$

$Owns(Nono,M1)$ $\quad\quad\quad\quad$ $Missile(M1)$

$Missile(x) \wedge Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$

$\neg Missile(x) \vee \neg Owns(Nono,x) \vee Sells(West,x,Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$\neg Missile(x) \vee Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$\neg Enemy(x, America) \vee Hostile(x)$

$American(West)$

Example2 for CNF- Conversion

Right: Textual representation
Below: FOL representation
Below right: CNF representation

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$
B. $\forall x\ [\exists z\ Animal(z) \wedge Kills(x,z)] \Rightarrow [\forall y\ \neg Loves(y,x)]$
C. $\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$
D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
E. $Cat(Tuna)$
F. $\forall x\ Cat(x) \Rightarrow Animal(x)$
¬G. $\neg Kills(Curiosity,Tuna)$

A1. $Animal(F(x)) \vee Loves(G(x),x)$
A2. $\neg Loves(x,F(x)) \vee Loves(G(x),x)$
B. $\neg Loves(y,x) \vee \neg Animal(z) \vee \neg Kills(x,z)$
C. $\neg Animal(x) \vee Loves(Jack,x)$
D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
E. $Cat(Tuna)$
F. $\neg Cat(x) \vee Animal(x)$
¬G. $\neg Kills(Curiosity,Tuna)$

Resolution Rule

- In propositional logic, resolution requires two disjunctions with complementary literals, i.e. one *is* the negation of the other
- In first-order logic, resolution requires two disjunctions with complementary literals, i.e. one *unifies* with the negation of the other:

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

with Unify (l_i, $\neg m_j$) = θ

- Example: We can resolve the two clauses:

  [Animal (F(x)) $\vee$ Loves (G(x), x)] and [$\neg$Loves (u,v) $\vee$ $\neg$Kills(u,v)] to

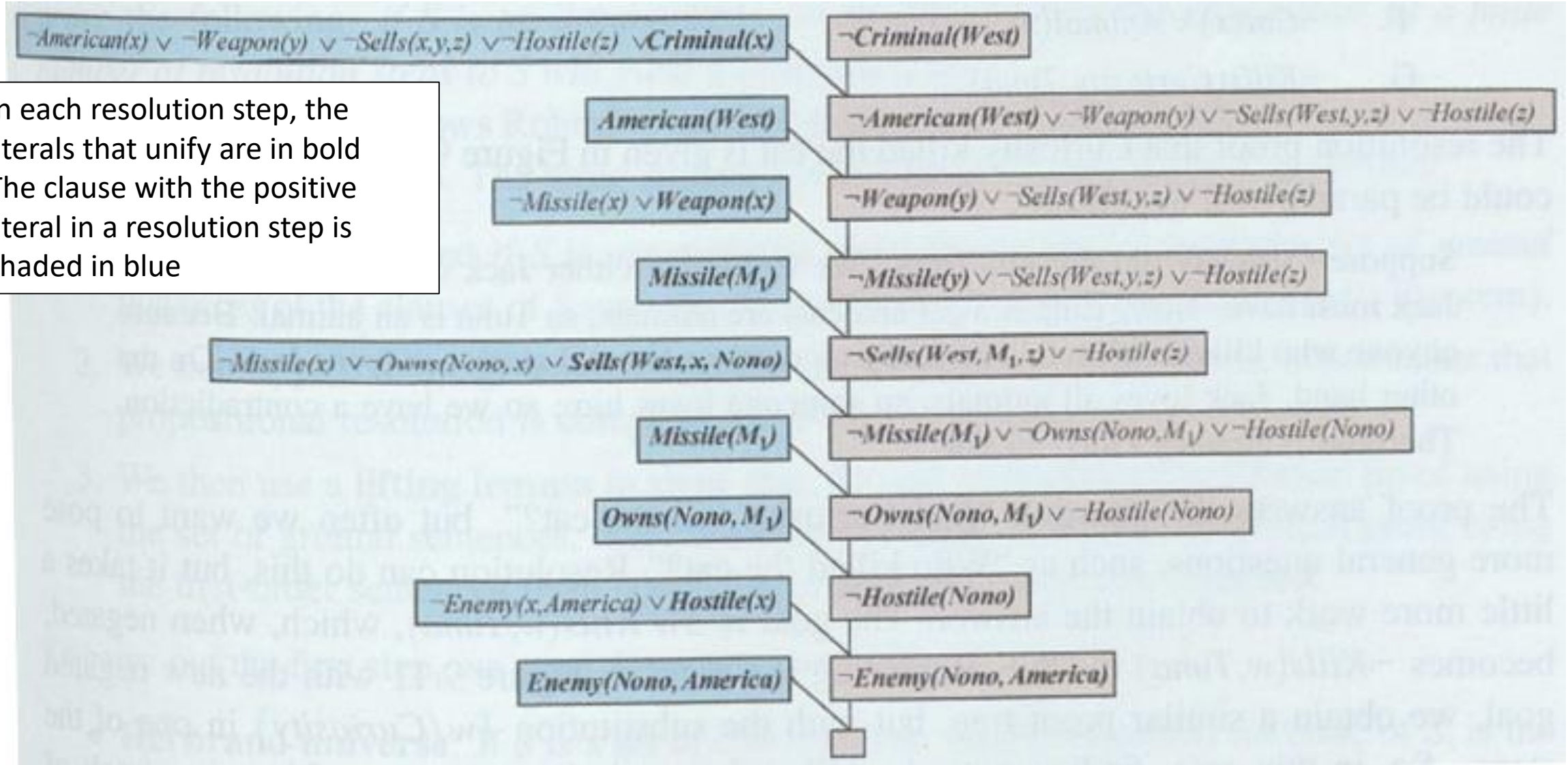
  [Animal (F(x)) $\vee$ $\neg$Kills(G(x),x)] with the unifier

  θ = {u/G(x), v/x}

Example Proof 1 based on CNF representation

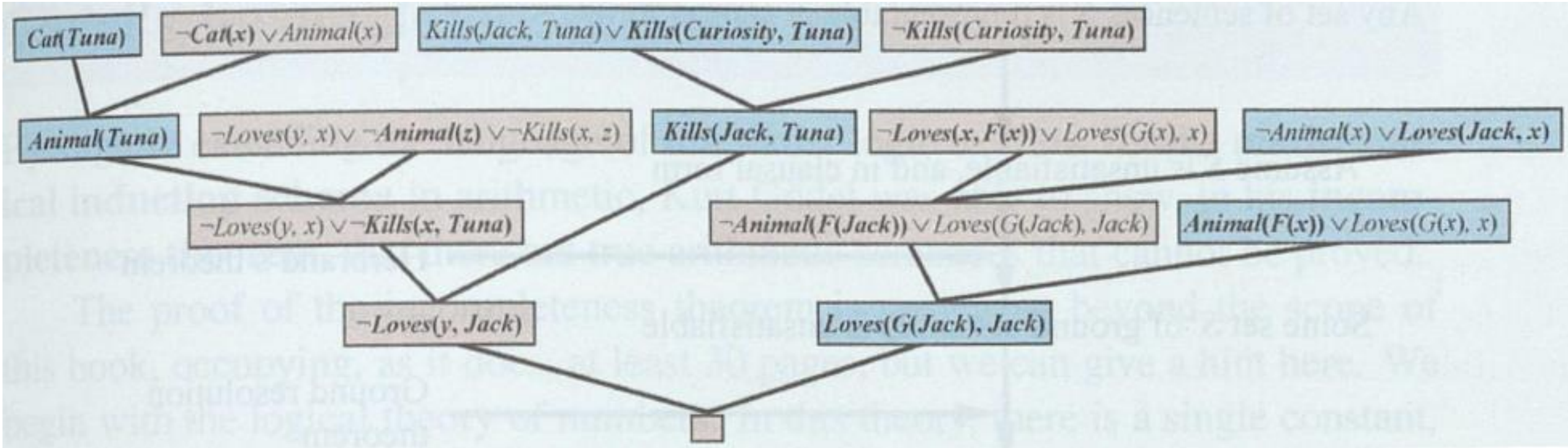
To prove Criminal (West), ¬Criminal (West) is added to show a refutation (i.e. empty clause)

- In each resolution step, the literals that unify are in bold
- The clause with the positive literal in a resolution step is shaded in blue



Example Proof 2 based on CNF representation

- To prove Kills (Curiosity, Tuna) $\neg$ *Kills(Curiosity, Tuna)* is added to the clauses to show a refutation (i.e. empty clause)
- In each resolution step, the literals that unify are in bold
- The clause with the positive literal in a resolution step is shaded in blue

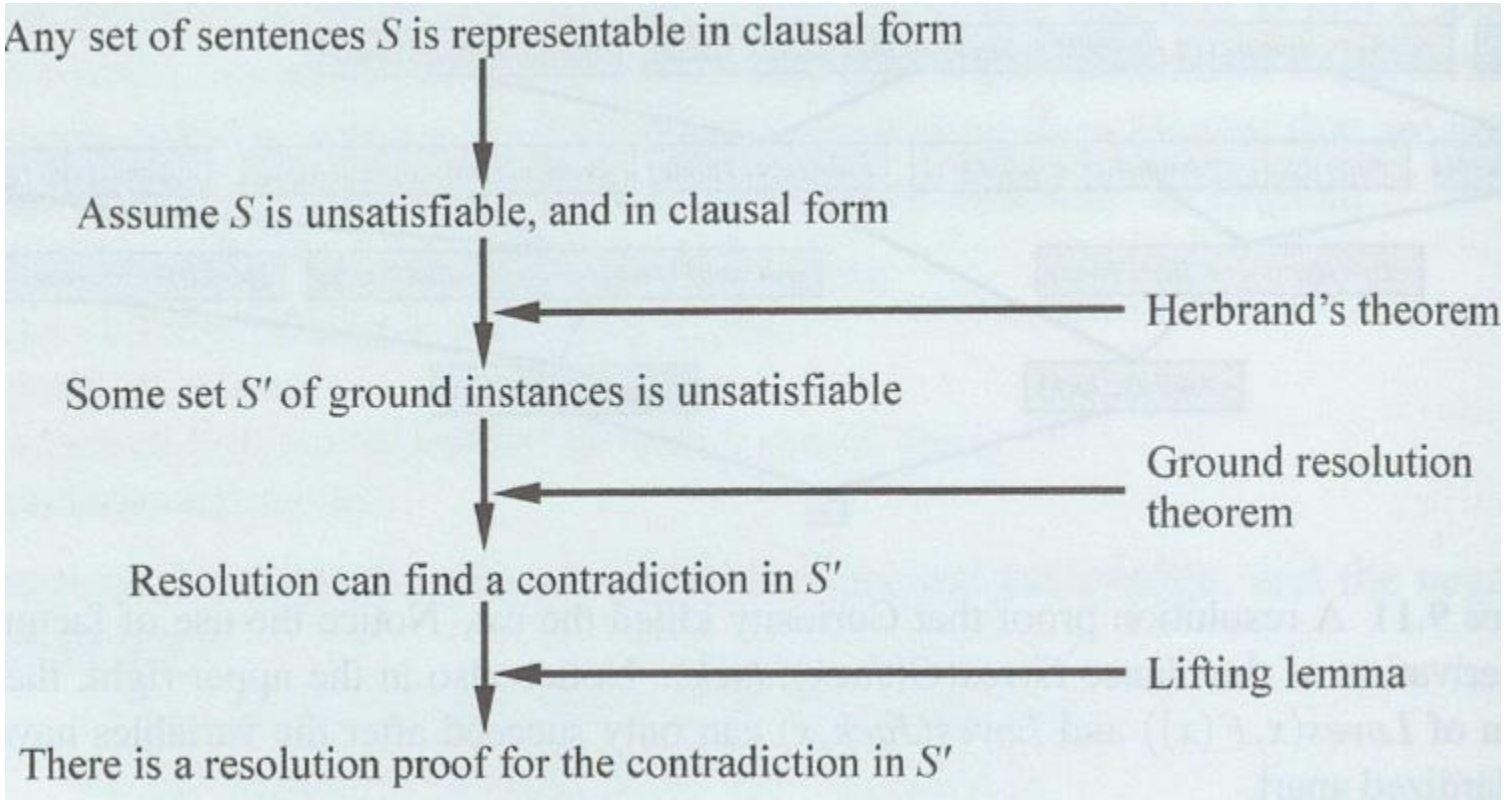


Completeness of Resolution

Resolution is refutation-complete, which means, that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction.

The proof idea is, that …

1. for each refutation for clauses S with variables, there is a set of ground instances of the clauses S such that this set in also unsatisfiables (Herbrand‘s theorem)
2. propositional resolution is complete
3. from propositional resolution proof, a corresponding first order resolution proof exist using the first-order sentences from which the ground sentences were obtained



Equality

- Till now, we cannot handle assertions of the form x = y
- Three distinct approaches can be taken:
  - Demodulation
  - Paramodulation
  - Equational unification
- Increases complexity of resolution resp. unification substantially

Resolution Strategies

How can resolution made efficient?

- **Unit preference:** Prefer sentences with a single literal (incomplete)
- **Set of support:** Use only sentences from the query and their resolvents (complete, goal-directed)
- **Input resolution:** Combine one of the input sentences (from KB or query) with some other sentence (incomplete)
  - **Linear resolution:** Generalization of input resolution: P and Q may be selected for resolution, if P is in the original KB or an ancestor of Q in the proof tree (complete)
- **Subsumption:** Eliminate all sentences subsumed by an existing sentence in the KB
- **Learning:** Given a collection of previously proved theorem, learn from them (e.g. find proof step similar to successful proof steps in the past).

Applications

- Logic have been successful in scenarios involving formal, strictly defined concepts, such as synthesis and verification of both hardware and software
- A theorem prover uses resolution strategies (see last slide) and optionally further knowledge (e.g. may use equations only in one direction as rewrite rules, e.g. x+0=x)
- However: Most complex real world scenarios have too much uncertainty and too many unknowns