<u>Course</u>: Introduction to AI

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 9

Inference in First-Order Logic

Outline

- Propositional vs. First-Order Inference
  - Inference rules for quantifiers
  - Reduction to propositional inference
- Unification and Lifting
  - A first-order inference rule
  - Unification
- Forward Chaining
  - First-order definite clauses
  - A simple forward-chaining algorithm
  - Efficient forward chaining

Outline

- Backward Chaining
  - A backward-chaining algorithm
  - Logic programming
  - Efficient implementation of logic programs
  - Redundant inference and infinite loops
  - Database semantics of Prolog
  - Constraint logic programming
- Resolution
  - Conjunctive normal form for first-order logic
  - The resolution inference rule
  - Completeness of resolution
  - Equality
  - Resolution strategies

Propositional vs. FO Inference

- In this chapter, we extend the results of the previous chapters to obtain algorithms that can answer any answerable question stated in FOL.
- We begin with simple inference rules to remove quantifiers from sentences.
- ➔ FO inference can be done by converting the KB to *PL* and using *propositional* inference.
- **<u>Inference rules for quantifiers:</u>**
- <u>Universal Quantifiers:</u>
- Suppose the axiom stating that all greedy kings are evil:

  $\forall x \; King(x) \land Greedy(x) \Rightarrow Evil(x)$
- This is clearly meant to apply to any object in the domain!

Propositional vs. FO Inference

- So it is permissible to infer any of the following sentences:

  King(John) ∧ Greedy(John) ⇒ Evil(John)

  King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)

  King(Father(John)) ∧ Greedy(Father(John)) ⇒ Evil(Father(John))

  …
- Rule of **Universal Instantiation** (**UI**) says that we can infer any sentence obtained by substituting a **ground term** for the variable.
- Let $SUBST(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written $\dfrac{\forall v\ \alpha}{SUBST(v/g, \alpha)}$

  for any variable v and ground term g.
- E.g. the previous 3 sentences are obtained by applying the substitutions: *{x/John}, {x/Richard }, and {x/Father(John)}.*

Propositional vs. FO Inference

- Rule for **Existential Instantiation**: the variable is replaced by a single *new constant symbol*.
- More formally: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base, $\dfrac{\exists\, v\; \alpha}{SUBST(v/k, \alpha)}$
- For example, from the sentence

  ∃x Crown(x) ∧ OnHead(x, John)
- We can infer the following sentence, C1 not appearing in KB

  Crown(C1) ∧ OnHead(C1, John)
- In logic, the new constant name is called a **Skolem constant.**
- The new KB is not logically equivalent to the old, but it is **inferentially equivalent**, i.e. that it is satisfiable exactly when the original KB is satisfiable.

Reduction to propositional inference

- Having removed the quantifiers, it is now possible to reduce first-order inference to propositional inference.
- The **first idea**: just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations.
- E.g., suppose the KB contains just the sentences:

  $\forall x \;\; King(x) \land Greedy(x) \Rightarrow Evil(x)$

  King(John).

  Greedy(John).

  Brother(Richard, John).
- Apply UI to the first sentence using all possible ground-term substitutions from the KB vocabulary (so *{x/John} {x/Richard}*)
  We obtain $King(John) \land Greedy(John) \Rightarrow Evil(John)$

  $King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard)$

Reduction to propositional inference

- Now, the KB is essentially propositional if the ground atomic sentences—*King(John), Greedy(John)*, and so on— are viewed as proposition symbols.
- So, one can apply any of the complete propositional algorithms in Chapter 7 to obtain conclusions such as Evil(John).
- This technique of **propositionalisation** can be made completely general, as shown further in this chapter.
- That is, every first-order KB and query can be propositionalised in such a way that entailment is preserved.

Unification and Lifting

- Propositionalisation is outdated (1960s) and inefficient.
- E.g. from the previous KB, it does not make sense to generate a sentence like:

  $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$

  while the obvious conclusion is Evil(John).
- To do an inference using $\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ we need to find a substitution $\theta = \{x/value\}$, where *value* is in the KB such that θ applied to the implication would produce the conclusion *Evil(value)*
- Suppose that instead of knowing *Greedy(John)* we know that $\forall y\ Greedy(y)$.
- We would still like to be able to conclude that *Evil(John)*, because we know that John is a *King (given)* and *John* is greedy (because everyone is greedy).
- We need the substitution *{x/John, y/John}*

Unification and Lifting

- This inference process can be captured as a single inference rule called **Generalized Modus Ponens**: For atomic sentences $p_{i,}\ p_i'$ and q , where there is a substitution θ such that $SUBST(\theta_,\ p_i') = SUBST(\theta_,\ p_i\ )$, for all *i*,

$$\frac{p_1', p_2', \ldots, p_{n,}'\quad (p_1 \wedge p_2 \wedge \ldots \wedge\ p_n\ \Rightarrow\ q)}{SUBST(\theta_, q)}$$

- *For our example:*

p_1' *is King(John)*	*p1 is King(x)*
p_2' *is Greedy(y)*	*p2 is Greedy(x)*
θ is {x/John, y/John}	*q is Evil(x)*

SUBST(θ, q) is Evil(John)

- It is easy to show that Generalized Modus Ponens is a sound inference rule.
- Generalized MP is a **lifted** version of MP—it raises MP from ground PL to FOL.

Unification

- Lifted inference rules require finding substitutions that make different logical expressions look identical.
- This process is called **unification** and is a key component of all FO inference algorithms.
- The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$UNIFY(p,q) = \theta \text{ where } SUBST(\theta,p) = SUBST(\theta,q)$$

- Consider the query: whom does John know?

$$AskVars(Knows(John,x))$$

- Results of unification with four sentences that might be in the KB:

1. *UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}*
2. *UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}*
3. *UNIFY(Knows(John, x), Knows(y,Mother (y))) = {y/John, x/Mother(John)}*
4. *UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail .*

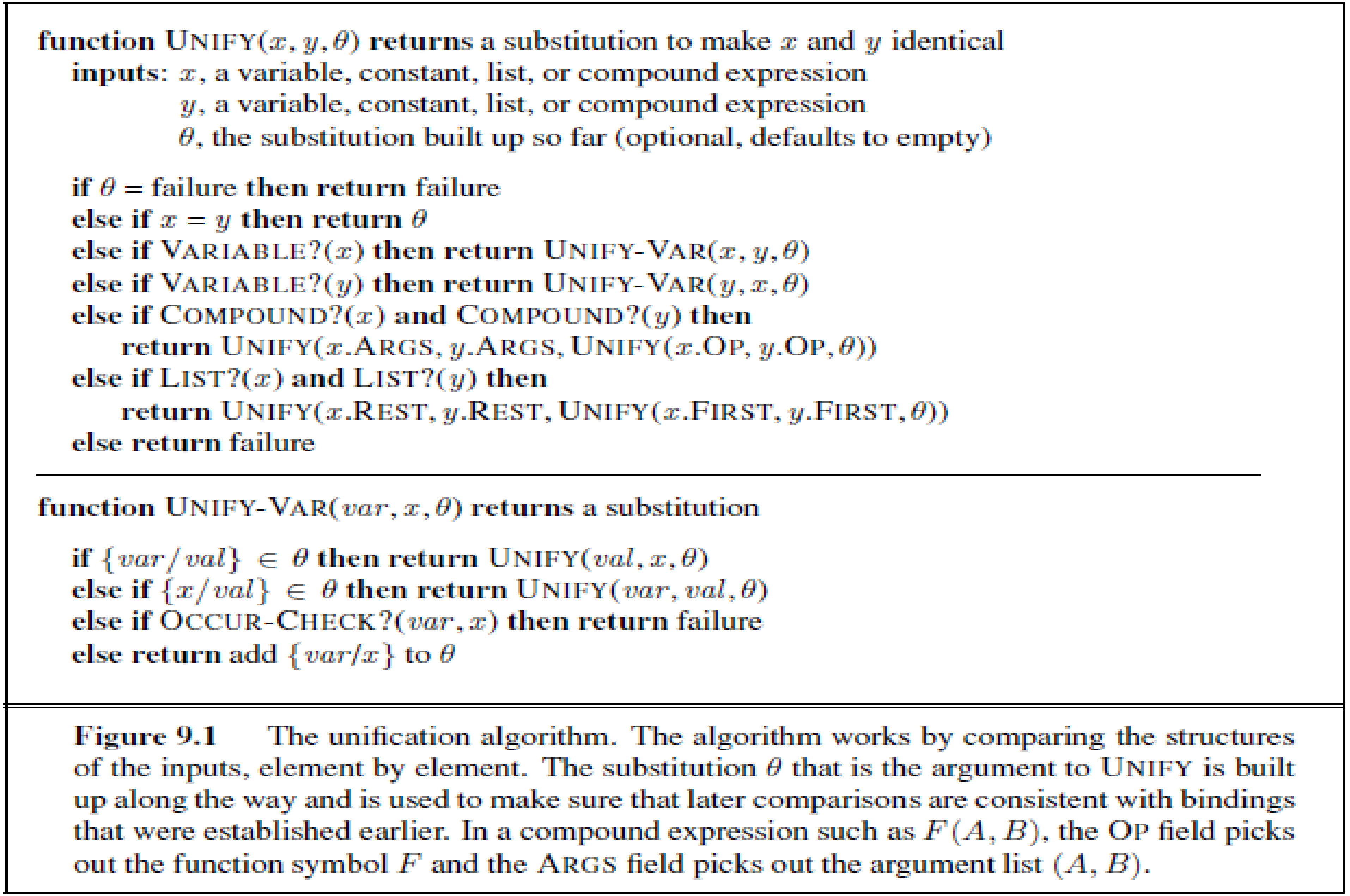
Unification

- **One complication**: the sentence *Knows(x, Elizabeth)* means *Everybody knows Elizabeth* ➔ including John.
- So unification should succeed, not fail!
- The problem is with the same variable x in the 2 sentences which have different meanings.
- ➔ Need to **standardize apart** one of the two sentences being unified, i.e. rename its variables to avoid name clashes.
- E.g. rename *x* in *Knows(x, Elizabeth)* to *y* (or any other new variable name) without changing its meaning.
- Now the unification will work: $UNIFY(\mathrm{Knows}(\mathrm{John}, \mathrm{x}), \mathrm{Knows}(\mathrm{y}, \mathrm{Elizabeth})) = \{\mathbf{y/John, x/Elizabeth}\}$
- **Other complication**: there could be more than one unifier.
- UNIFY(Knows(John, x), Knows(y, z)) could return {y/John, x/z} or {y/John, x/John, z/John}.

Unification

- The first unifier gives *Knows(John, z)* as the result of unification, whereas the second gives *Knows(John, John)*.
- The second result could be obtained from the first by an additional substitution {z/John};
- The first unifier is said to be *more general* than the second: it places fewer restrictions on the values of the variables.
- General rule: *for every unifiable pair of expressions, there is a single* ***most general unifier*** *(MGU) that is unique up to a renaming and substitution of variables*.
- Example: {x/John} and {y/John} are considered equivalent, as are {x/John, y/John} and {x/John, y/x}.)
- In the previous case, the MGU is {y/John, x/z}.

Unification Algorithm



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
   inputs: $x$, a variable, constant, list, or compound expression
           $y$, a variable, constant, list, or compound expression
           $\theta$, the substitution built up so far (optional, defaults to empty)

   if $\theta$ = failure then return failure
   else if $x = y$ then return $\theta$
   else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
   else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
   else if COMPOUND?($x$) and COMPOUND?($y$) then
       return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
   else if LIST?($x$) and LIST?($y$) then
       return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
   else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

   if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
   else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
   else if OCCUR-CHECK?($var, x$) then return failure
   else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

- Unification is simple: recursively explore the two expressions "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match.
- One expensive step: when matching a variable against a complex term, must check if the variable occurs inside the term;
- if yes, the match fails: no consistent unifier can be constructed.
- E.g., S(x) can't unify with S(S(x)).
- This is called **occur check:**
  It prevents variable instantiation that leads to an infinite loop; **BUT** it makes the complexity of the unification algorithm quadratic in the size of the expressions being unified.
- Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result;
- Other systems use more complex algorithms with linear-time complexity.

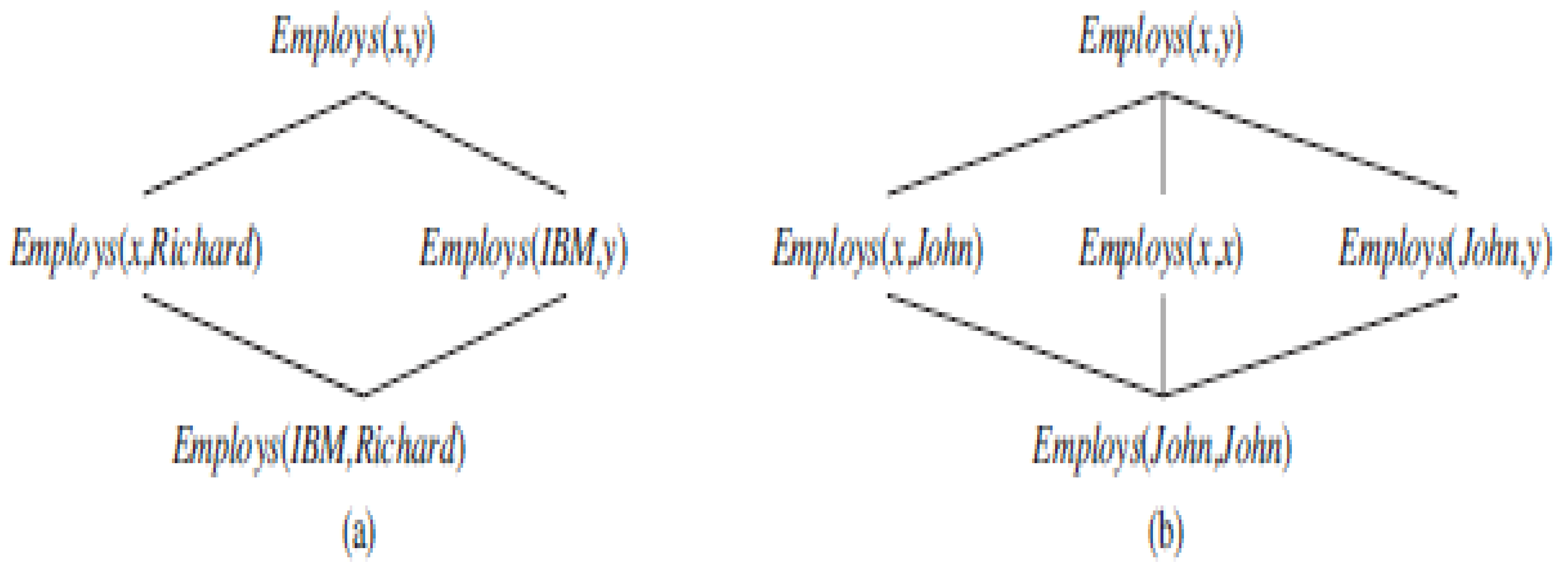


Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM*, *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

Forward Chaining

- A forward-chaining algorithm for propositional definite clauses was given (Chapter 7).
- <u>Simple idea</u>: start with the atomic sentences in the KB and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.
- **First-order definite clauses:** As for propositional definite clauses, they are disjunctions of literals of which *exactly one is positive*. Examples:

  King(x) ∧ Greedy(x) ⇒ Evil(x) .

  King(John) .

  Greedy(y) .
- Any variables are assumed to be universally quantified.

Forward Chaining

- Consider the following (NL) text:

The law 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, write the NL sentence in FOL.
- ". . . *it is a crime for an American to sell weapons to hostile nations*": Rule **(R1)**

American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)

- "*Nono . . . has some missiles.*" sentence *∃x Owns(Nono, x) ∧ Missile(x)* is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:

  Owns(Nono, M1) **(R2)**

  Missile(M1) **(R3)**

Forward Chaining

- *"All of its missiles were sold to it by Colonel West":*

  $Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$ **(R4)**
- <u>It is known that </u>missiles are weapons:

  $Missile(x) \Rightarrow Weapon(x)$ **(R5)**
- *<u>It is known that </u>*an enemy of America counts as "hostile":

  $Enemy(x, America) \Rightarrow Hostile(x)$ **(R6)**
- "*West, who is American . . .":*

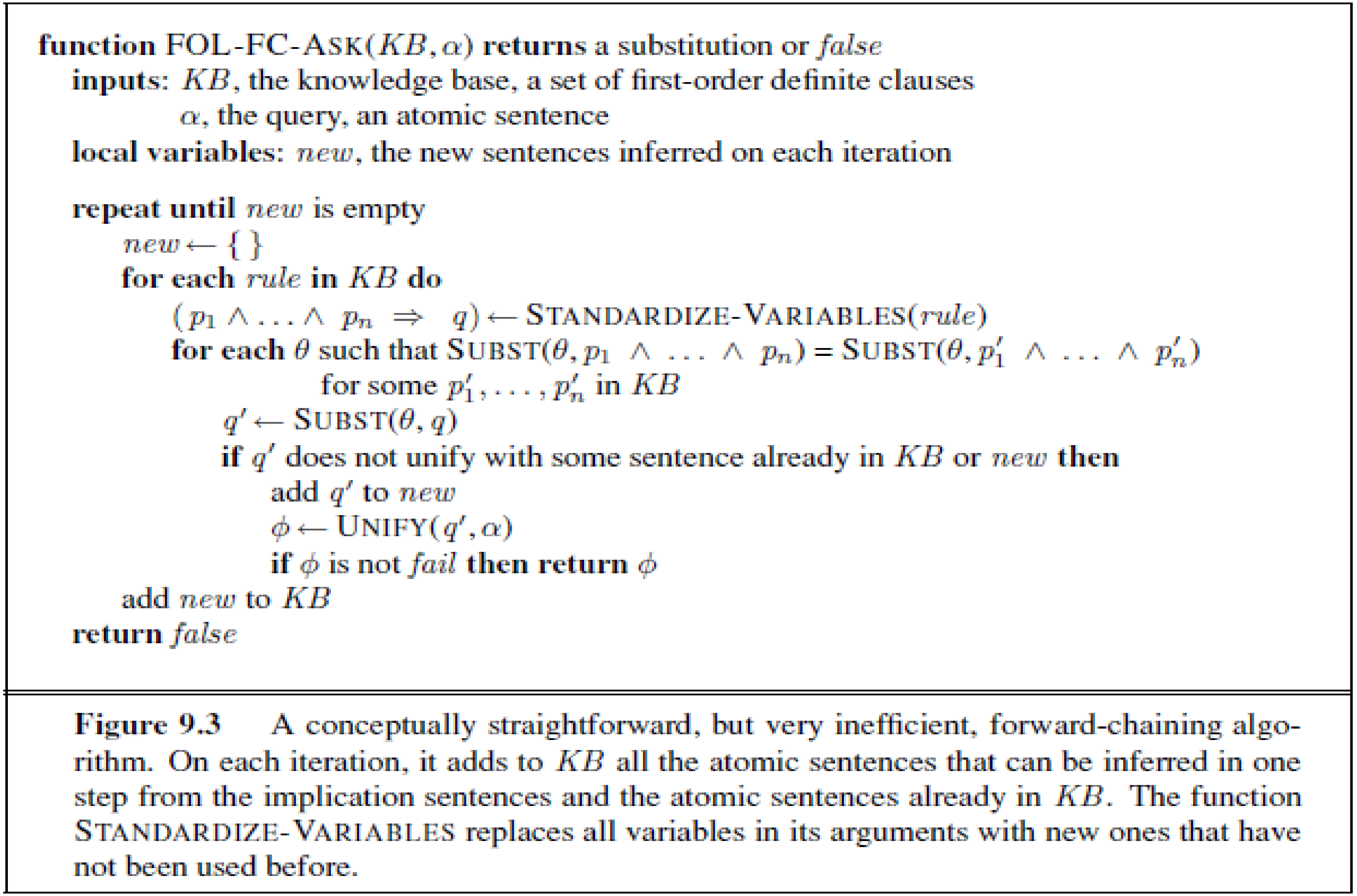
  $American(West)$ **(R7)**
- *"The country Nono, an enemy of America . . .":*

  $Enemy(Nono, America)$ **(R8)**
- This KB is a **Datalog** KB, i.e. a KB which uses no function.
- In the absence of function symbols, inference is much easier.

A simple forward-chaining algorithm

- This FC algorithm
  - starts from the known facts,
  - triggers all the rules whose premises are satisfied,
  - adds their conclusions to the known facts.
  - The process repeats until
    - the query is answered (assuming that just one answer is required) or
    - no new facts are added.
- Notice that a fact is not “new” if it is just a **renaming** of a known fact.
- E.g., Likes(x, IceCream) and Likes(y, IceCream) are renamings of each other.
- The FOL-FC-ASK algorithm does the above simple FC.

A simple forward-chaining algorithm



```
function FOL-FC-ASK($KB, \alpha$) returns a substitution or $false$
  inputs: $KB$, the knowledge base, a set of first-order definite clauses
          $\alpha$, the query, an atomic sentence
  local variables: $new$, the new sentences inferred on each iteration

  repeat until $new$ is empty
      $new \leftarrow \{\}$
      for each $rule$ in $KB$ do
          $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
          for each $\theta$ 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($\theta, q$)
              if $q'$ does not unify with some sentence already in $KB$ or $new$ then
                  add $q'$ to $new$
                  $\phi \leftarrow$ UNIFY($q', \alpha$)
                  if $\phi$ is not $fail$ then return $\phi$
      add $new$ to $KB$
  return $false$
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

Simple FC algorithm with Crime Pb.

- The crime problem is used to illustrate how FOL-FC-ASK works.
- The implication sentences are (R1), (R4), (R5), and (R6). Two iterations are required:
- On the first iteration:
  - Rule (R1) has unsatisfied premises.
  - Rule (R4) is satisfied with *{x/M1}* ➔ *Sells(West,M1, Nono)* is added.
  - Rule (R5) is satisfied with *{x/M1}*, and *Weapon(M1)* is added.
  - Rule (R6) is satisfied with *{x/Nono}* ➔ *Hostile(Nono)* is added.
- On the second iteration:
  - Rule (R1) is satisfied with *{x/West, y/M1, z/Nono}* ➔ *Criminal (West)* is added.

Simple FC algorithm with Crime Pb.

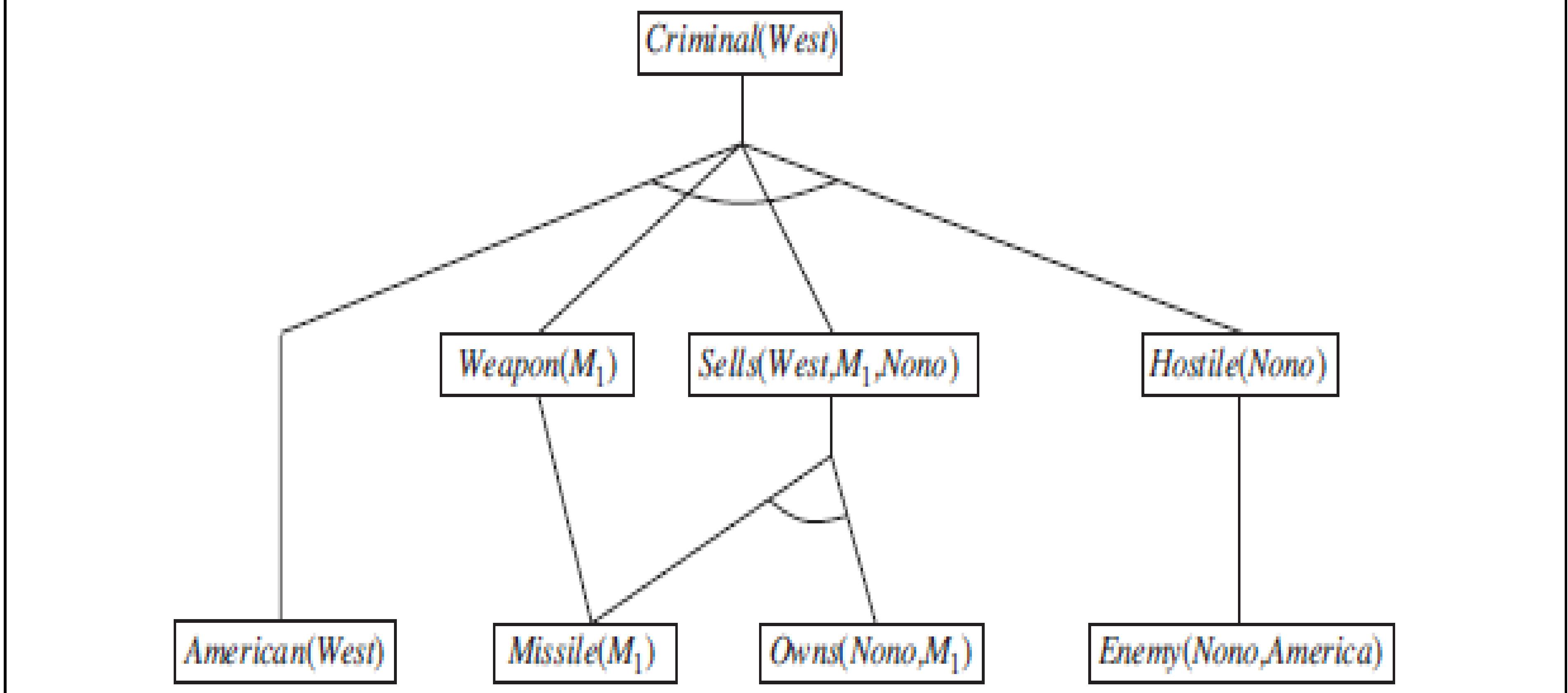


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Properties of FOL-FC-ASK

- Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB.
- Such a KB is called a **fixed point** of the inference process.
- FOL-FC-ASK is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound.
- It is **complete** for definite clause KBs.
- For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts. E.g.:

  NatNum(0)

  $\forall n\ NatNum(n) \Rightarrow NatNum(S(n))$,
- Then forward chaining adds NatNum(S(0)), NatNum(S(S(0))), NatNum(S(S(S(0)))), and so on.

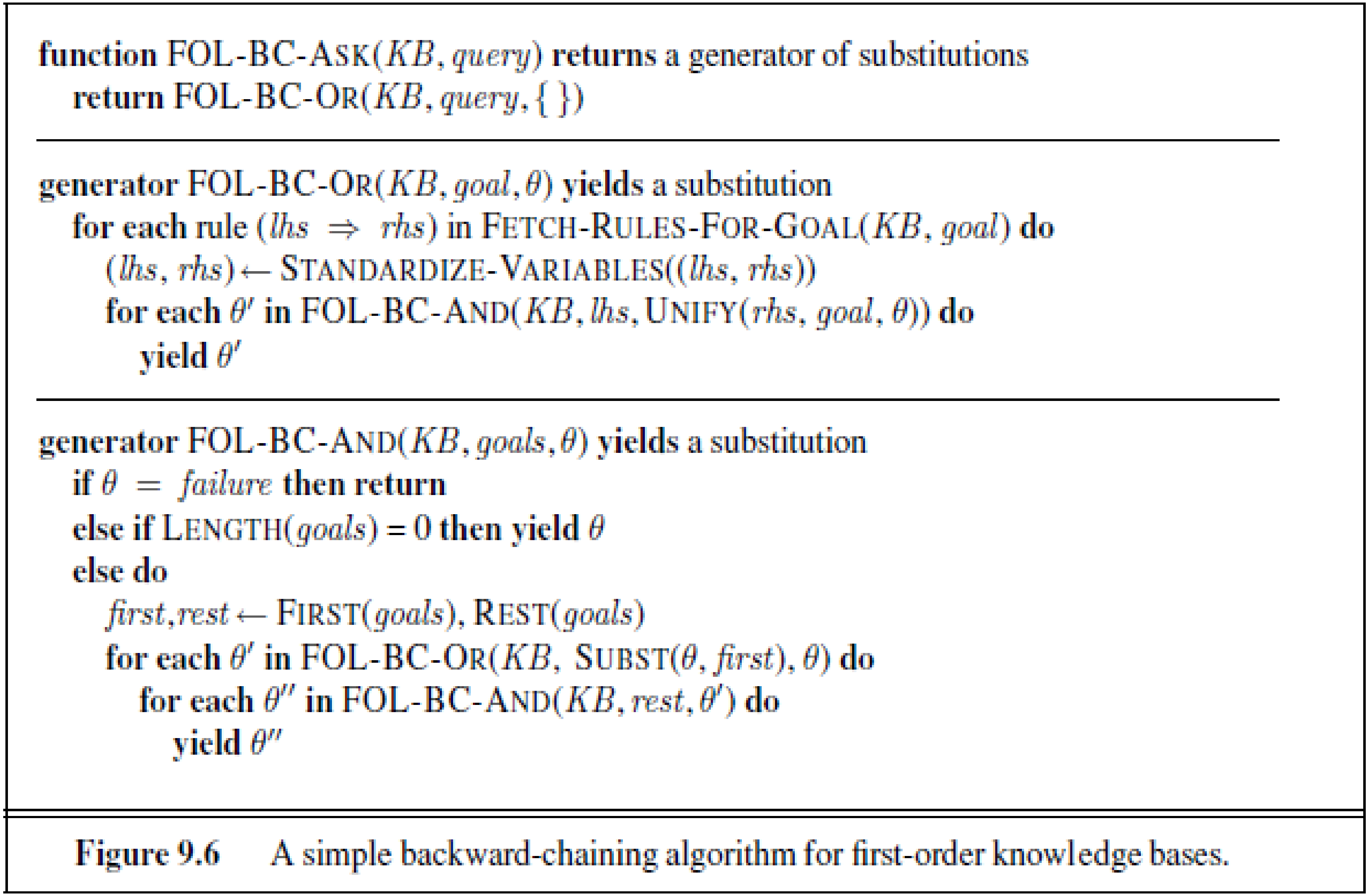
On the inefficiency of FOL-FC-ASK

- The FOL-FC-ASK algorithm is designed for ease of understanding rather than for efficiency of operation.
- There are three possible sources of inefficiency.
    1. Its "inner loop" involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This can be very expensive.
    2. The algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the KB on each iteration.
    3. The algorithm might generate many facts that are irrelevant to the goal.
- Efficient solutions have been introduced and implemented in actual systems to alleviate these inefficiencies.

Backward Chaining

- The **Backward Chaining** algorithm is a DFS algorithm which works backward from the goal, chaining through rules to find facts that support the proof.
- *FOL-BC-ASK(KB, goal)* will be proved if the KB contains a clause of the form *lhs ⇒ goal*, where *lhs* is a list of conjuncts and these can be proved to be true.
- An atomic fact like *American(West)* is considered as a clause whose *lhs* is the empty list.
- A query that contains variables might be proved in multiple ways.
- E.g., the query *Person(x)* could be proved with the substitution *{x/John}* as well as with *{x/Richard}.*
- BC is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the *lhs* of a clause must be proved.

Backward Chaining



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

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    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″
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

Proof tree by Backward Chaining

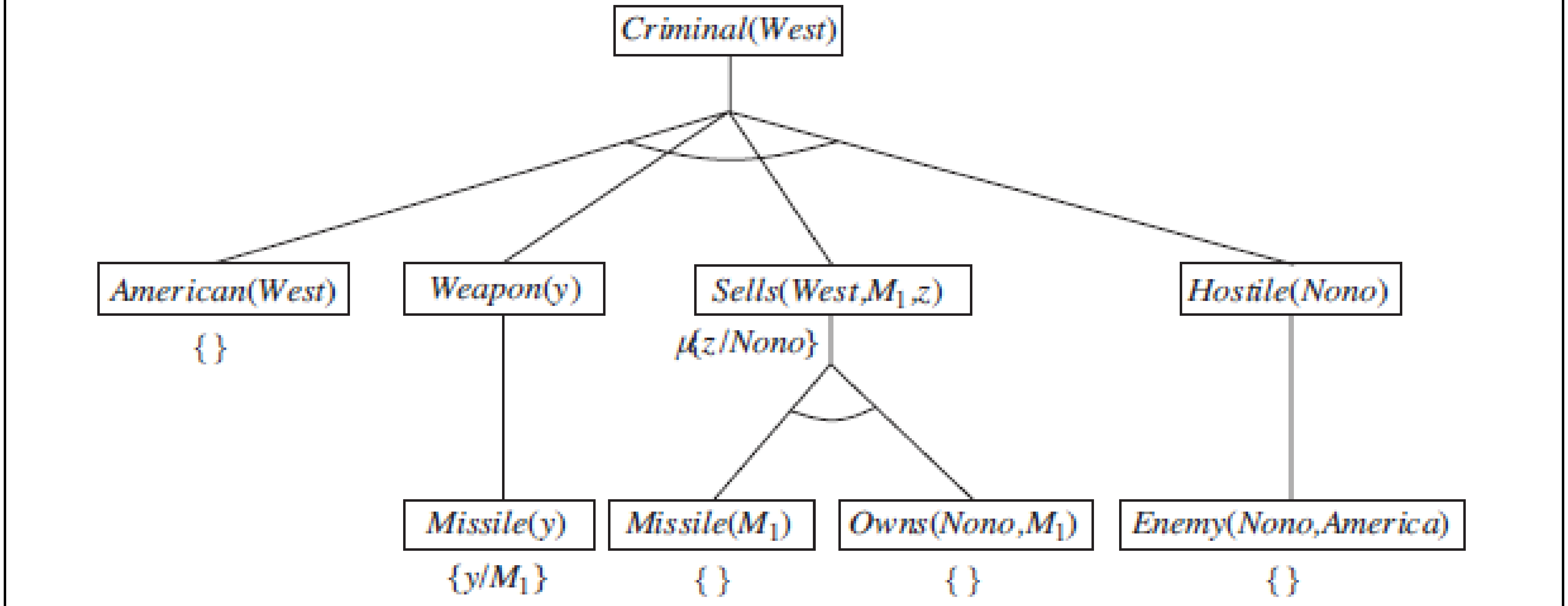


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile*(z), z is already bound to *Nono*.

Logic programming and Prolog

- Logic programming is a technology that comes fairly close to embodying the declarative ideal.
- The ideal is summed up in Robert Kowalski's equation:

  Algorithm = Logic + Control
- **Prolog** is the most widely used logic programming language.
- Prolog programs are sets of definite clauses written in a notation somewhat different from standard FOL.
- The syntax and conventions are different. E.g.

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

- Example of list concatenation *(append):*

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

Prolog

- The Prolog definition, e.g. of *append*, is much more powerful than procedural languages implementation: Prolog describes a *relation* that holds among three arguments, rather than a *function* computed from two arguments.
- So one can ask a query like *append(X,Y,[1,2])!*
- We get back the solutions

  X=[] Y=[1,2];

  X=[1] Y=[2];

  X=[1,2] Y=[]
- Prolog applies DFS with leftmost literal first.
- Clauses are tried in the order in which they are written in the KB.

Prolog

Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics seen in Chapter 8 rather than first-order semantics.
- There is a set of built-in functions for arithmetic. Literals using these function symbols are “proved” by executing code rather than doing further inference.
  - E.g., the goal “X is 4+3” succeeds with X bound to 7.
  - On the other hand, the goal “5 is X+Y” fails, because the built-in functions do not do arbitrary equation solving.
- There are built-in predicates that have side effects when executed. These include
  - input–output predicates and
  - the assert/retract predicates for modifying the knowledge base.
- Such predicates have no counterpart in logic and can produce confusing results. E.g. if facts are asserted in a branch of the proof tree that eventually fails.

Prolog

- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- **<u>Exercise</u>**:Find an example of an unsound inference resulting from the non-application of the occur-check.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. ➔ it is <u>very fast</u> when given the right set of axioms, **but** <u>incomplete</u> when given the wrong ones.

Prolog’s incompleteness

- Prolog has a serious weakness: risk of infinite computations.
- Example: the following logic program that decides if a path exists between two points on a directed graph:

  path(X,Z) :- link(X,Z).

  path(X,Z) :- path(X,Y), link(Y,Z).

  link(a,b).

  link(b,c).
- With this programme, the query *path(a,c)* succeeds.
- If the order of the first two rules is reversed:
- The query *path(a,c)* gets into an infinite loop.
- This example shows that Prolog is **incomplete** as a theorem prover for definite clauses—even for Datalog programs.
- Forward chaining does not suffer from this problem: once *path(a,b), path(b,c),* and *path(a,c)* are inferred, FC halts.

Proof trees for the Path example

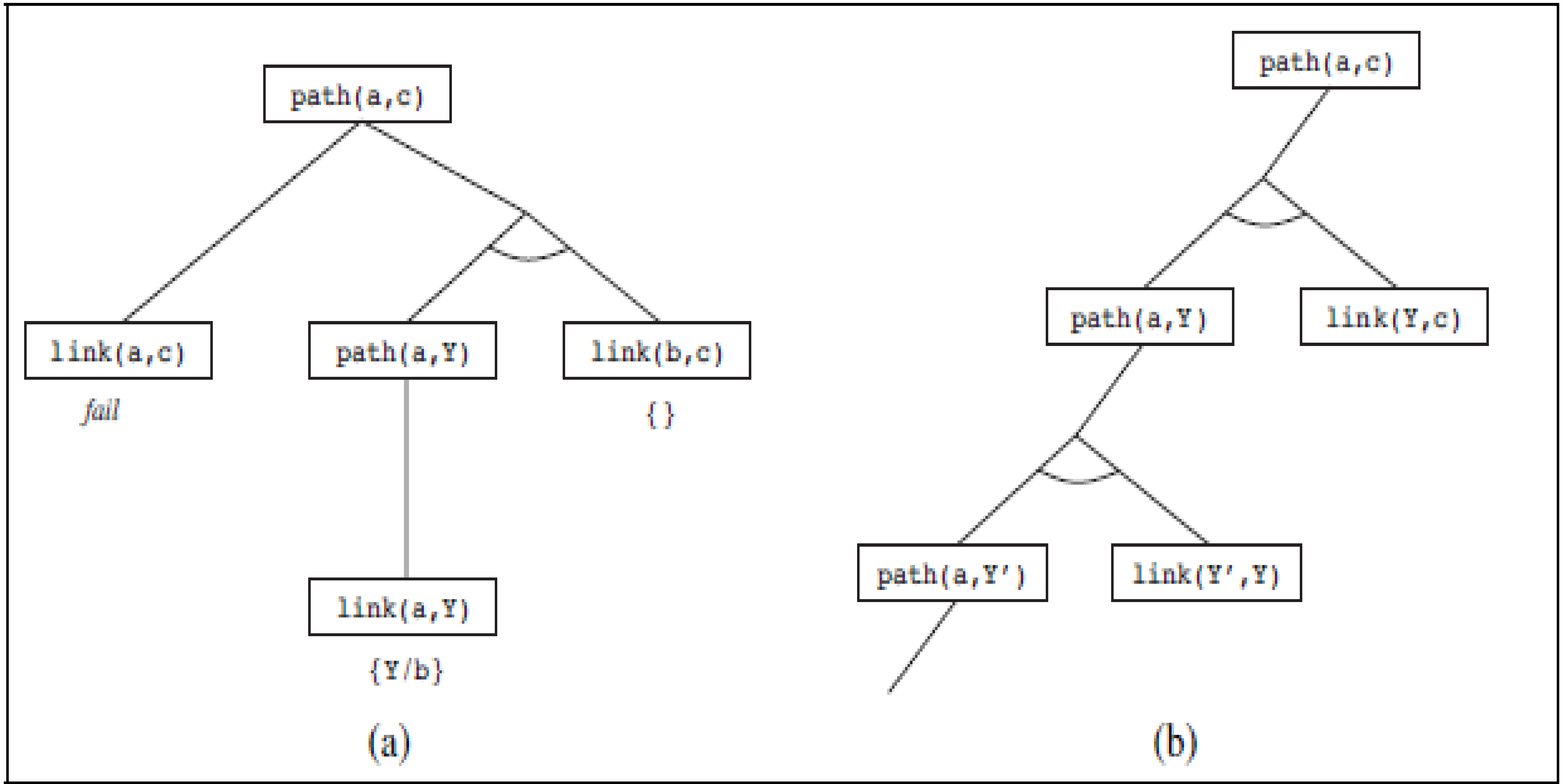


Figure 9.10 (a) Proof that a path exists from *A* to *C*. (b) Infinite proof tree generated when the clauses are in the “wrong” order.

Resolution

- As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF).
- Literals can contain variables, which are assumed to be universally quantified.

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

becomes, in CNF:

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

- *Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.*
- We will first explain Skolemisation.

Skolemisation

- **Skolemisation** is the process of removing existential quantifiers by elimination.
  - In the simple case, it is just like the Existential Instantiation rule : translate $\exists x\ P(x)$ into $P(A)$, where A is a new constant.
  - When the existentially quantified variable v is in the scope of some universally quantified variables, introduce a Skolem function to replace v where the function has as arguments all <u>these</u> universally quantified variables.
  - E.g. Skolemising the sentence:

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

gives

$$\forall x\ [Animal(F(x)) \land \neg Loves(x,F(x))] \lor Loves(G(x),x)$$

Skolemisation

- Other example:

  $\forall x \forall y\ [\neg P(x) \vee \neg S(y,x) \vee \exists z\ [A(z)$ ^

  $W(x,z)$ ^ $R(y,z)]]$

becomes

  $\forall x \forall y\ [\neg P(x) \vee \neg S(y,x) \vee [A(H(x,y))$ ^

  $W(x, H(x,y))$ ^ $R(y, H(x,y))]]$

Conversion of FOL sentences to CNF

- The procedure for conversion to CNF is similar to the propositional case.
- Principal difference: the need to eliminate existential quantifiers.
- Example: *"Everyone who loves all animals is loved by someone"*. In FOL, this sentence is:

  $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$
- The steps to convert to CNF are as follows:
- **Eliminate implications**:

First step:

$\forall x\ [\neg(\forall y\ \text{Animal}(y) \Rightarrow \text{Loves}(x,y))] \vee [\exists y\ \text{Loves}(y,x)]$

- **Move $\neg$ inwards**:

$\neg\forall x\ p$	becomes	$\exists x\ \neg p$
$\neg\exists x\ p$	becomes	$\forall x\ \neg p$

Conversion of FOL sentences to CNF

Second step:

$$\forall x\ [\ \exists y\ \neg(\ Animal(y) \Rightarrow Loves(x,y))]\ \vee\ [\exists y\ Loves(y,x)]$$

- The sentence goes through the following transformations:

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

- **Standardize variables**: For sentences like $(\exists x\ P(x)) \vee (\exists x\ Q(x))$, rename the occurrence of *x* in one of sentences.
- This avoids confusion later when quantifiers are dropped. Hence the sentence:

Conversion of FOL sentences to CNF

- **Standardize variables:**

  $\forall x\ [\exists y\ Animal(y) \land \neg Loves(x,y)] \lor [\exists z\ Loves(z,x)]$
- **Skolemize:** The $\exists$ are in the scope of $\forall x$ so:

  $\forall x\ [Animal(F(x)) \land \neg Loves(x,F(x))] \lor Loves(G(x),x)$

  where F and G are **Skolem functions**
- **Drop universal quantifiers**:

  $[Animal(F(x)) \land \neg Loves(x,F(x))] \lor Loves(G(x),x)$.
- **Distribute $\lor$ over $\land$:**

  $[Animal(F(x)) \lor Loves(G(x),x)] \land$

  $[\neg Loves(x,F(x)) \lor Loves(G(x),x)]$
- The sentence is now in CNF and consists of two clauses.

The resolution inference rule

- The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule of Chapter 7.
- Two clauses, which are assumed to be standardized can be resolved if they contain complementary literals.
- Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies with* the negation of the other. So we have

$$\frac{l_1 \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_n}{SUBST(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $UNIFY(l_i, \neg m_j) = \theta$

- Example: resolve the two clauses [Animal (F(x)) ∨ Loves(G(x), x)] and [¬Loves(u, v) ∨ ¬Kills(u, v)]

with unifier θ={u/G(x), v/x}, to produce the **resolvent** clause

[Animal (F(x)) ∨ ¬Kills(G(x), x)]

On Resolution

- The previous rule is called the **binary resolution** rule because it resolves exactly two literals.
- The binary resolution rule is not a complete inference procedure.
- The full resolution rule resolves subsets of literals in each clause that are unifiable.
- We can also extend **factoring**—the removal of redundant literals—to the first-order case.
- Propositional factoring reduces two literals to one if they are *identical*; first-order factoring reduces two literals to one if they are *unifiable*. The unifier must be applied to the entire clause.
- The combination of binary resolution and factoring is complete.

Crime Example

- Recall: Resolution proves that $KB \models \alpha$ by proving $KB \wedge \neg\alpha$ unsatisfiable, i.e. by deriving empty clause

- *Crime example* sentences in CNF are

  ¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x, y, z) ∨ ¬Hostile(z) ∨ Criminal (x)

  ¬Missile(x) ∨ ¬Owns(Nono, x) ∨ Sells(West, x, Nono)

  ¬Enemy(x,America) ∨ Hostile(x)

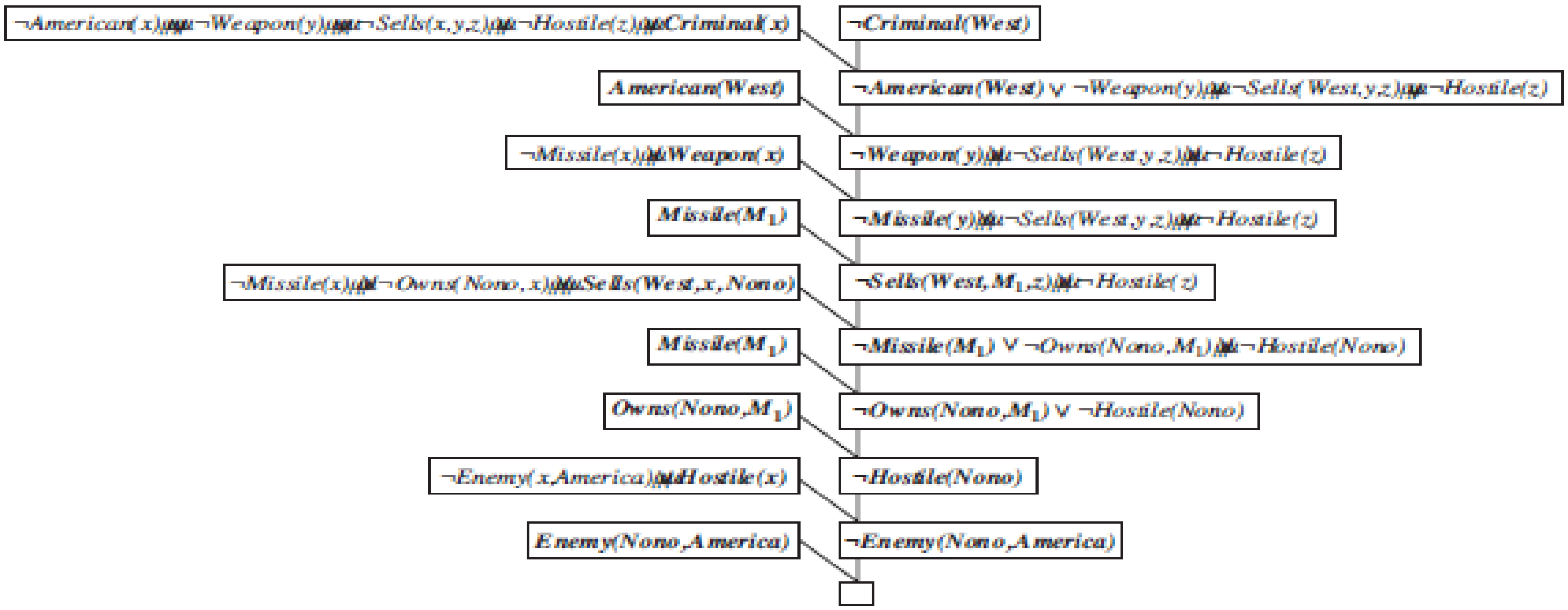
  ¬Missile(x) ∨Weapon(x)

  Owns(Nono,M1)

  Missile(M1)

  American(West)

  Enemy(Nono,America)



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

Figure 9.11 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

Second Proof Example: Who killed the cat?

- 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?

Second Proof Example: Who killed the cat?

- First, we express the original sentences, some background knowledge, and the negated goal *G* in FOL:
  - 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. $\forall x\ Cat(x) \Rightarrow Animal\ (x)$

- Negate the goal $\neg G$: $\neg Kills(Curiosity,Tuna)$

Second Proof Example: Who killed the cat?

- Converting each sentence to CNF yields:

A1. Animal(F(x)) v Loves(G(x), x)

A2. ¬Loves(x, F(x)) v Loves(G(x), x)

B. ¬Loves(y, x) v ¬Animal (z) v ¬Kills(x, z)

C. ¬Animal(x) v Loves(Jack, x)

D. Kills(Jack, Tuna) v Kills(Curiosity, Tuna)

E. Cat(Tuna)

F. ¬Cat(x) v Animal (x)

¬G: ¬Kills(Curiosity, Tuna)

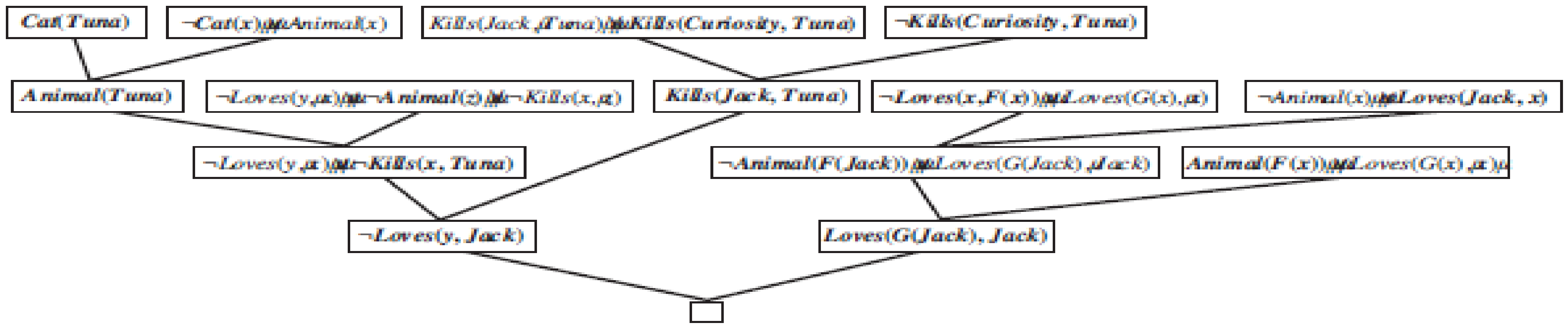

Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y, x) ∨ ¬Animal(z) ∨ ¬Kills(x, z)
Kills(Jack, Tuna)
¬Loves(x, F(x)) ∨ Loves(G(x), x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y, x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack), Jack)
Animal(F(x)) ∨ Loves(G(x), x)
¬Loves(y, Jack)
Loves(G(Jack), Jack)

Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$. Notice also in the upper right, the unification of $Loves(x, F(x))$ and $Loves(Jack, x)$ can only succeed after the variables have been standardized apart.

Worded proof

- In English, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna.

We know that either Jack or Curiosity did; thus Jack must have.

Now, Tuna is a cat and cats are animals, so Tuna is an animal.

Because anyone who kills an animal is loved by no one, we know that no one loves Jack.

On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction.

Therefore, Curiosity killed the cat.

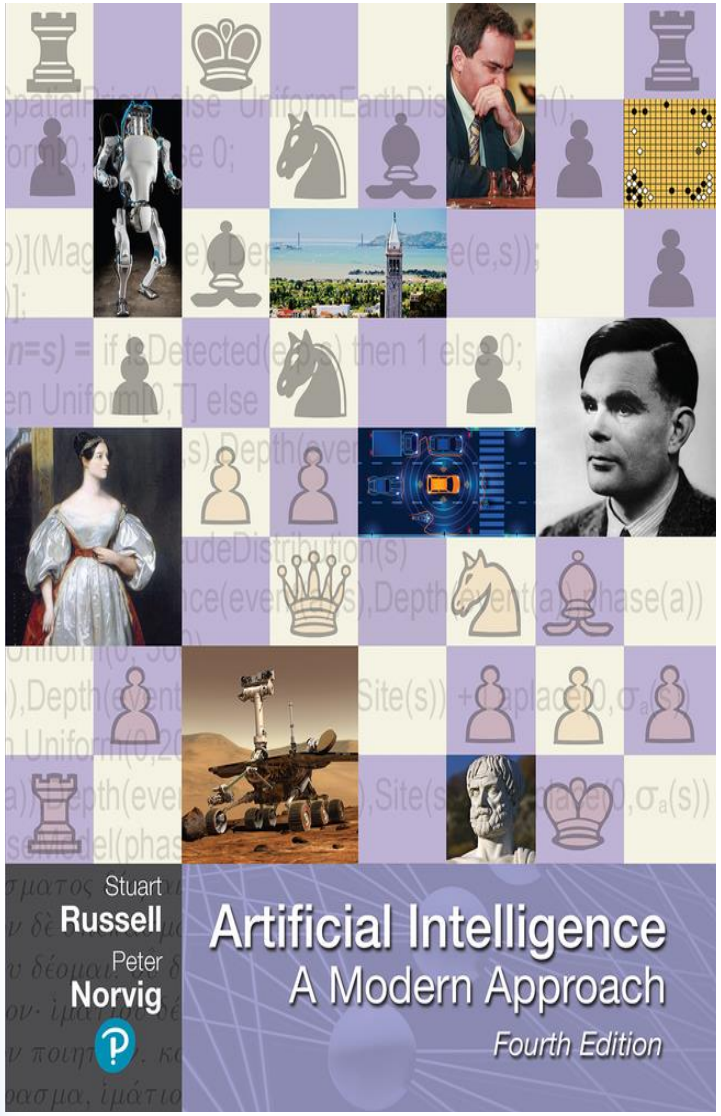
Resolution and proofs

- The proof answers the question "*Did Curiosity kill the cat?*"
- We may want to pose more general questions, e.g. "*Who killed the cat?*"
- Resolution can do this.
- The goal is *∃w Kills(w, Tuna)*, which, when negated, becomes ¬Kills(w, Tuna) in CNF.
- Repeating the previous proof with the new negated goal, <u>we obtain a similar proof tree, but with the substitution *{w/Curiosity}* in one of the steps</u>.
- Finding out who killed the cat is just a matter of <u>keeping track of the bindings for the query variables</u> in the proof.

Completeness of resolution

- Resolution is **refutation-complete**, i.e. *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction.
- Resolution cannot be used to generate all logical consequences of a set of sentences, but <u>it can be used to establish that a given sentence is entailed by the set of sentences</u>.
- Hence, Resolution can be used to find all answers to a given question, Q(x), by proving that KB ∧ ¬Q(x) is unsatisfiable.

Slides based on the textbook



- Russel, S. and Norvig, P. (2020) Artificial Intelligence,
A Modern Approach (4th Edition), Pearson Education Limited.