Unification & Chaining

Outline

I. Unification

II. First-order definite clauses

III. Forward chaining

IV. Backward chaining

* Figures are from the textbook site.

I. Generalized Modus Ponens

$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$

Suppose there exists a substitution θ such that

$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i')$ for $1 \leq i \leq n$

Example $n = 1$, $p_1 \equiv$ *Dad*$(x,$ *John*$)$, $p_1' \equiv$ *Dad*$($*David*$, y)$, $\theta = \{x/$*David*, *y/John*$\}$

I. Generalized Modus Ponens

$$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$$

Suppose there exists a substitution θ such that

$$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$$

Example $n = 1$, $p_1 \equiv$ *Dad*$(x,$ *John*$)$, $p_1' \equiv$ *Dad*$($*David*$, y)$, $\theta = \{x/$*David*, *y/John*$\}$

Then

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

I. Generalized Modus Ponens

$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$

Suppose there exists a substitution θ such that

$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$

Example $n = 1$, $p_1 \equiv \textit{Dad}(x, \textit{John})$, $p_1' \equiv \textit{Dad}(\textit{David}, y)$, $\theta = \{x/\textit{David}, y/\textit{John}\}$

Then

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

KB:

$\textit{Gate}(X_1)$, $\textit{Terminal}(\textit{In}(1, C_1))$ $\qquad$ $\textit{Gate}(g) \wedge \textit{Terminal}(t) \Rightarrow g \neq t$

I. Generalized Modus Ponens

$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$

Suppose there exists a substitution θ such that

$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$

Example $n = 1$, $p_1 \equiv \textit{Dad}(x, \textit{John})$, $p_1' \equiv \textit{Dad}(\textit{David}, y)$, $\theta = \{x/\textit{David}, y/\textit{John}\}$

Then

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

KB:

$\textit{Gate}(X_1)$, $\textit{Terminal}(\textit{In}(1, C_1))$

$\textit{Gate}(g) \wedge \textit{Terminal}(t) \Rightarrow g \neq t$

$\text{SUBST}(\theta, q)$

$\theta = \{g/X_1, t/\textit{In}(1, C_1)\}$

q is $g \neq t$

I. Generalized Modus Ponens

$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$

Suppose there exists a substitution θ such that

$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$

Example $n = 1$, $p_1 \equiv \textit{Dad}(x, \textit{John})$, $p_1' \equiv \textit{Dad}(\textit{David}, y)$, $\theta = \{x/\textit{David}, y/\textit{John}\}$

Then

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

KB:

$\textit{Gate}(X_1)$, $\textit{Terminal}(\textit{In}(1, C_1))$ $\qquad$ $\textit{Gate}(g) \wedge \textit{Terminal}(t) \Rightarrow g \neq t$

$\text{SUBST}(\theta, q)$ ⇩ $\theta = \{g/X_1, t/\textit{In}(1, C_1)\}$, q is $g \neq t$

$X_1 \neq \textit{In}(1, C_1)$

Unification

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

Unification

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

sentences

Unification

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

sentences

Query: *AskVars*(*Knows*(*John*, x)) // what does John know?

Unification

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

sentences

Query: *AskVars*(*Knows*(*John*, x)) // what does John know?

Answers: all the sentences in the KB found to unify with *Knows*(*John*, x).

Unification

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

sentences

Query: *AskVars*(*Knows*(*John*, x)) // what does John know?

Answers: all the sentences in the KB found to unify with *Knows*(*John*, x).

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*)}

Unification (cont'd)

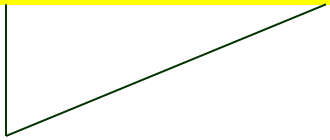
- Conflicting substitutions

$\text{UNIFY}(Knows(John, x), Knows(x, Elizabeth)) = failure$

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

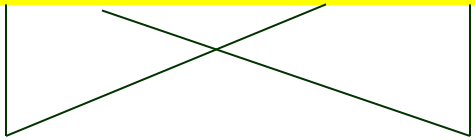


$\{x/John\}$

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

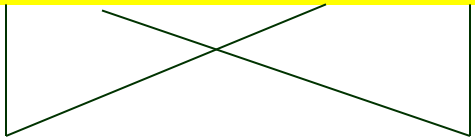


$\{x/John\}$ $\{x/Elizabeth\}$

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



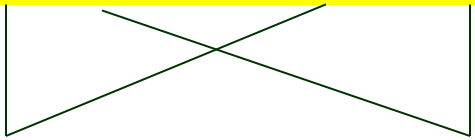
$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

UNIFY(*Knows*(*John*, x), *Knows*(y, z))

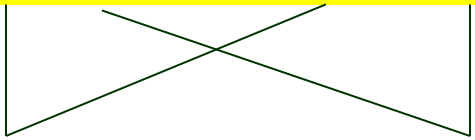
could return $\{y/John, x/z\}$

or $\{y/John, x/John, z/John\}$

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

UNIFY(*Knows*(*John*, x), *Knows*(y, z))

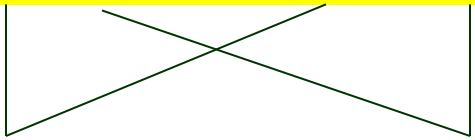
could return $\{y/John, x/z\}$ ⟹ *Knows*(*John*, z)

or $\{y/John, x/John, z/John\}$

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

UNIFY(*Knows*(*John*, x), *Knows*(y, z))

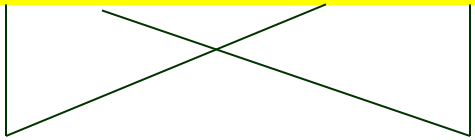
could return $\{y/John, x/z\}$ ⟹ *Knows*(*John*, z)

or $\{y/John, x/John, z/John\}$ ⟹ *Knows*(*John*, *John*)

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

UNIFY(*Knows*(*John*, x), *Knows*(y, z))

could return $\{y/John, x/z\}$ ⟹ *Knows*(*John*, z)

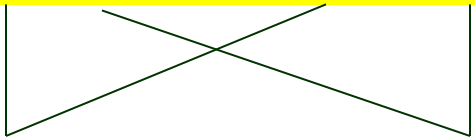
more general unifier for fewer restriction on variable values

or $\{y/John, x/John, z/John\}$ ⟹ *Knows*(*John*, *John*)

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

UNIFY(*Knows*(*John*, x), *Knows*(y, z))

could return $\{y/John, x/z\}$ ⟹ *Knows*(*John*, z)

more general unifier for fewer restriction on variable values

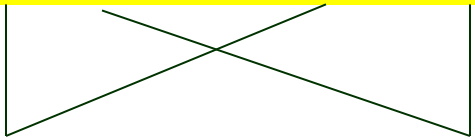
or $\{y/John, x/John, z/John\}$ ⟹ *Knows*(*John*, *John*)

less general unifier

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*



$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

UNIFY(*Knows*(*John*, x), *Knows*(y, z))

could return $\{y/John, x/z\}$ ⟹ *Knows*(*John*, z)

more general unifier for fewer restriction on variable values

or $\{y/John, x/John, z/John\}$ ⟹ *Knows*(*John*, *John*)

less general unifier

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 symbol of x

argument list of y

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ 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

Recursively explore two expressions x and y "side by side" to build up a unifier.

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 symbol of x

argument list of y

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure* // check whether the variable *var* appears
  else return add $\{var/x\}$ to θ // inside the complex term x. match fails if so
// because no unifier can be constructed.

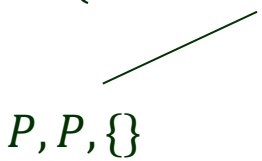
Recursively explore two expressions x and y "side by side" to build up a unifier.

Unification Example

UNIFY$\big(P(x, f(a, y), z), P\big(b, z, f(a, c)\big), \{\}\big)$ Variables: x, y, z

Unification Example

UNIFY$\big(P(x, f(a, y), z), P\big(b, z, f(a, c)\big), \{\}\big)$ Variables: x, y, z

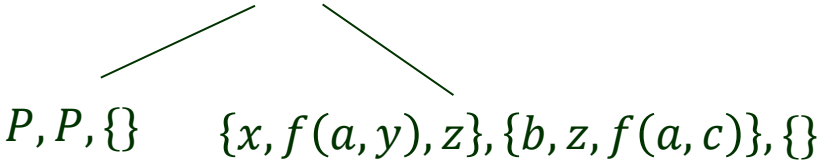


$P, P, \{\}$

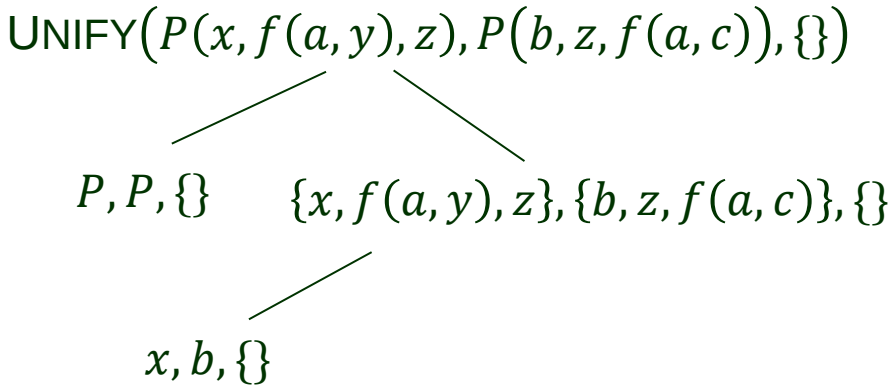
Unification Example

UNIFY$\big(P(x, f(a, y), z), P\big(b, z, f(a, c)\big), \{\}\big)$

Variables: x, y, z

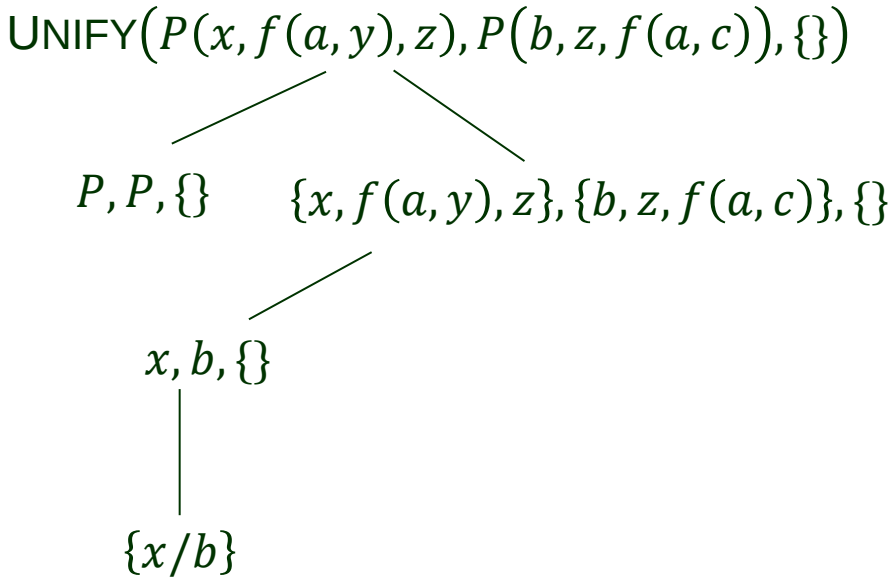


Unification Example



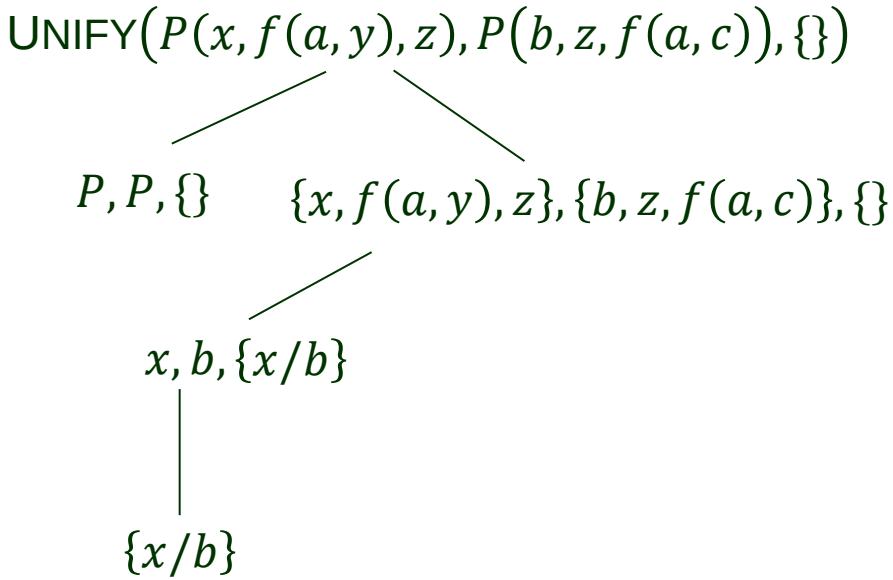
Variables: x, y, z

Unification Example



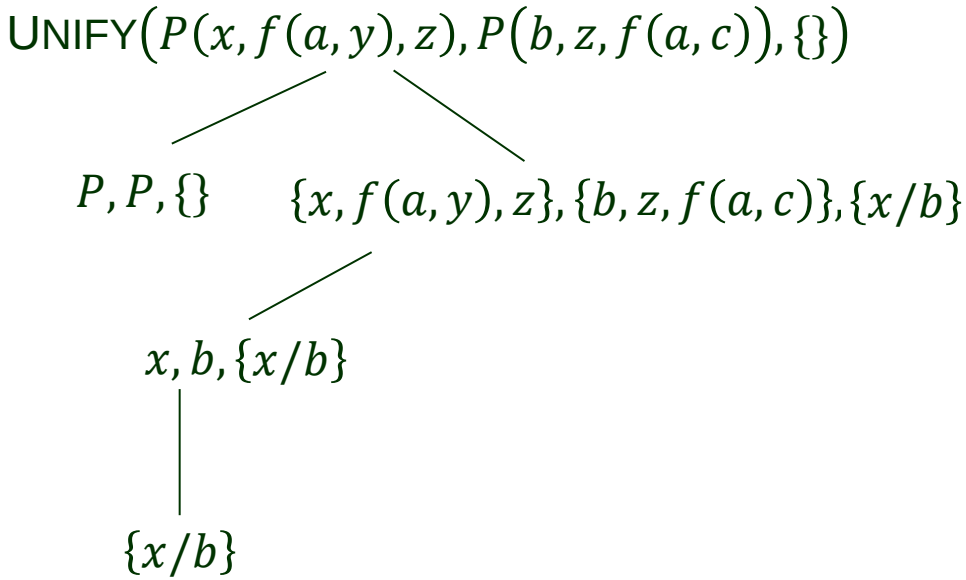
Variables: x, y, z

Unification Example

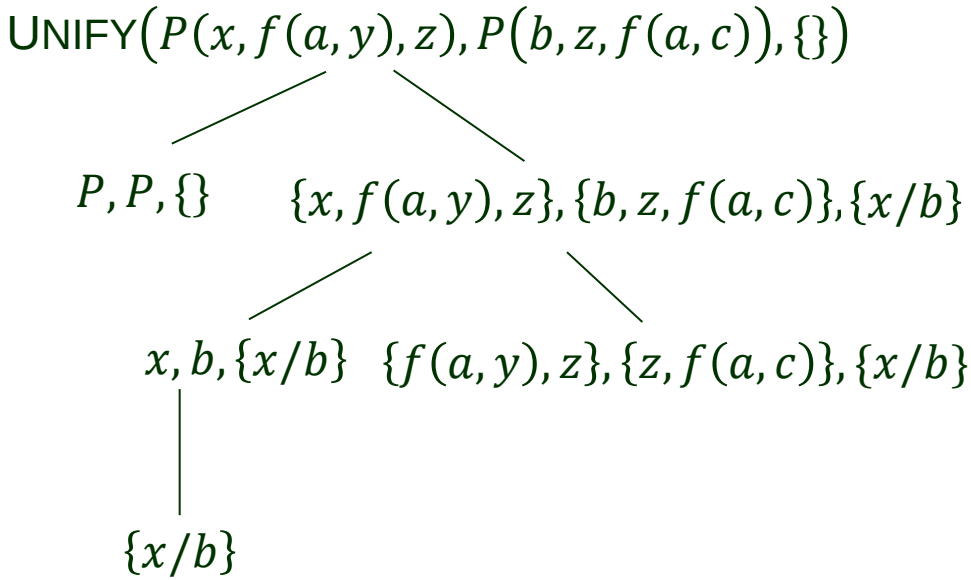


Variables: x, y, z

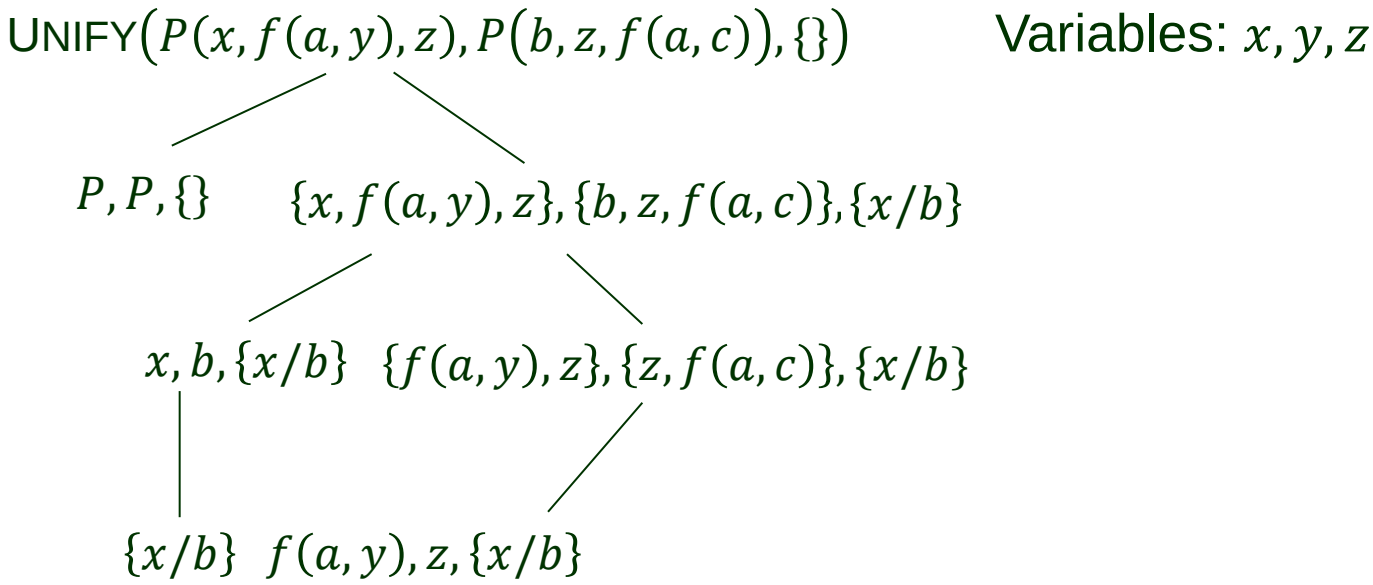
Unification Example



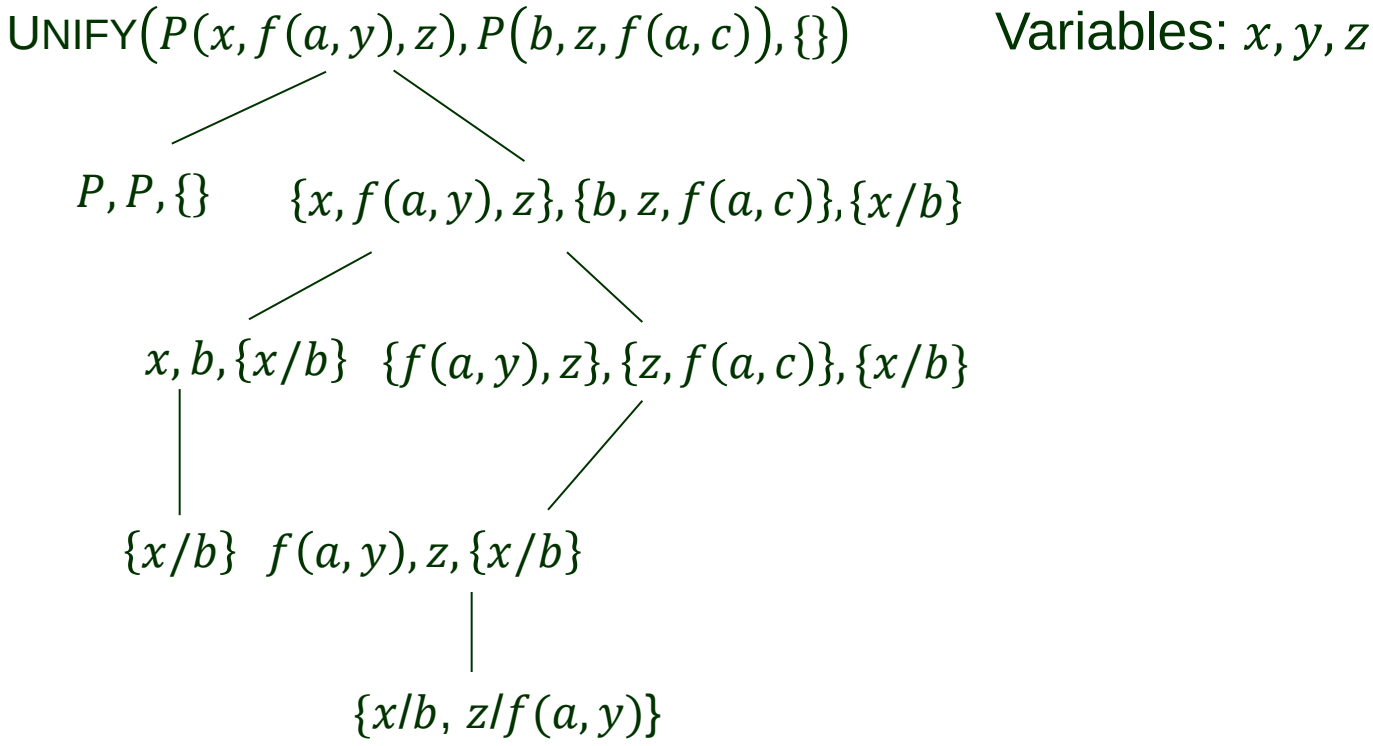
Unification Example



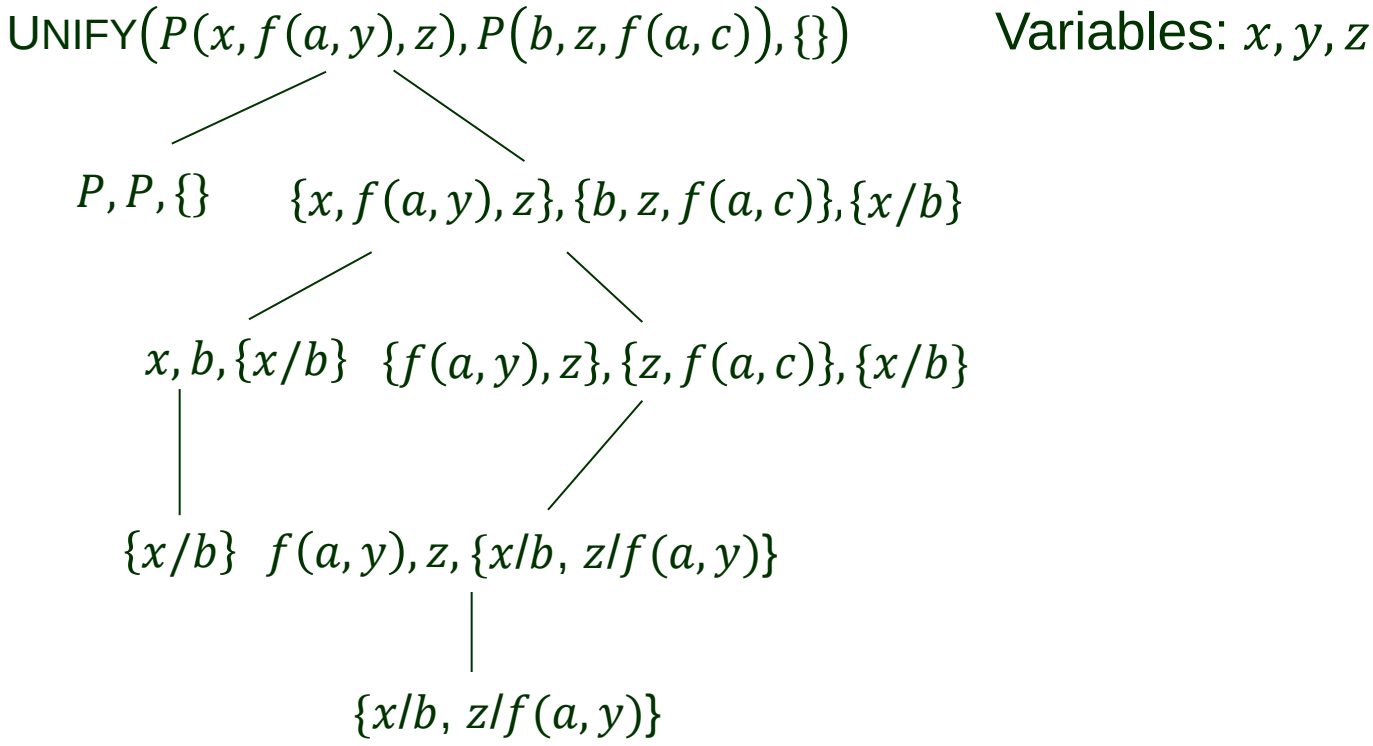
Unification Example



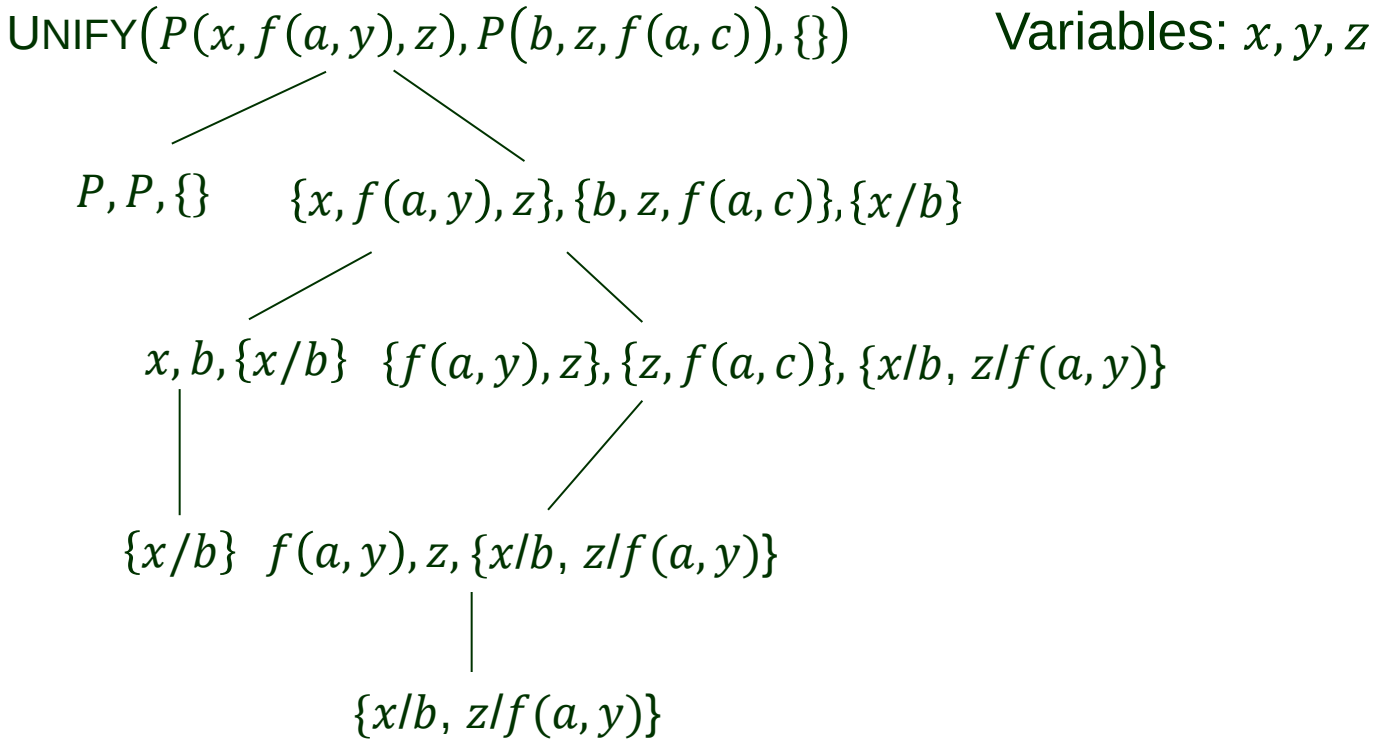
Unification Example



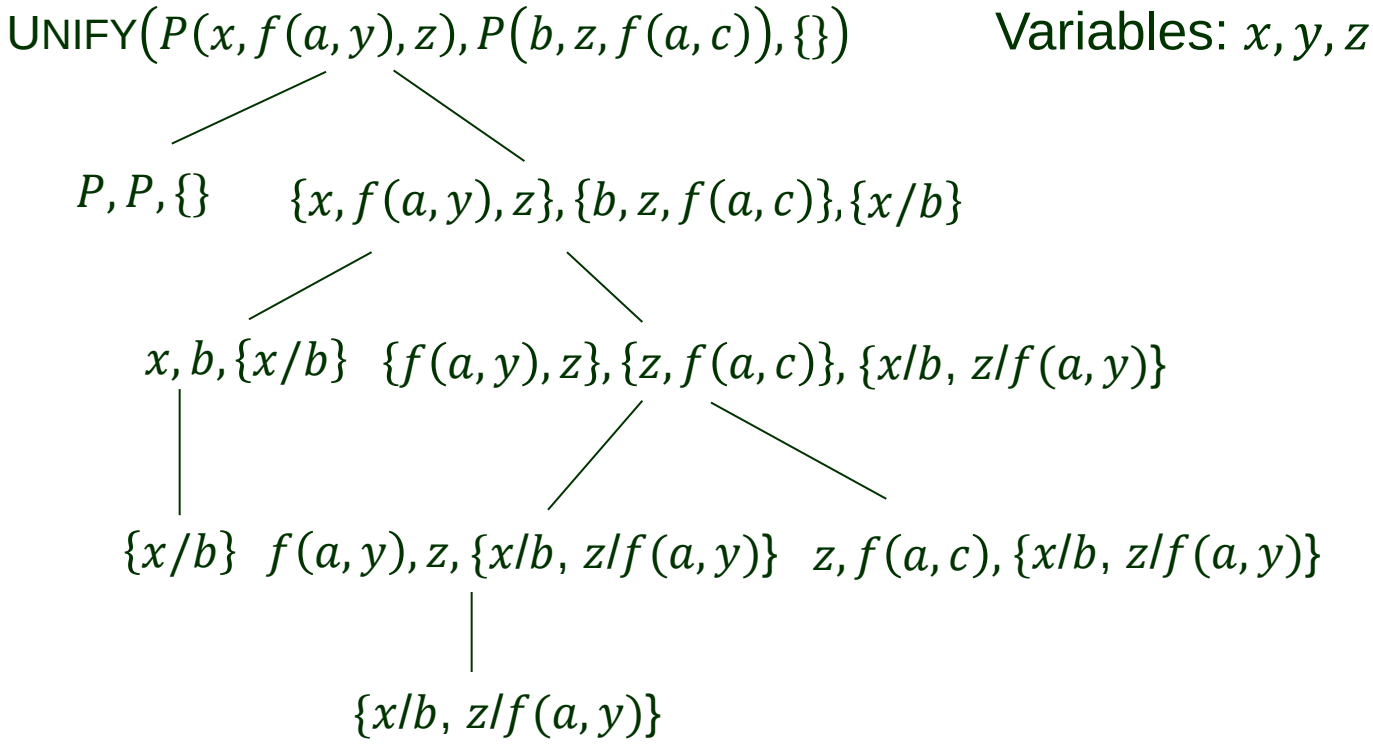
Unification Example



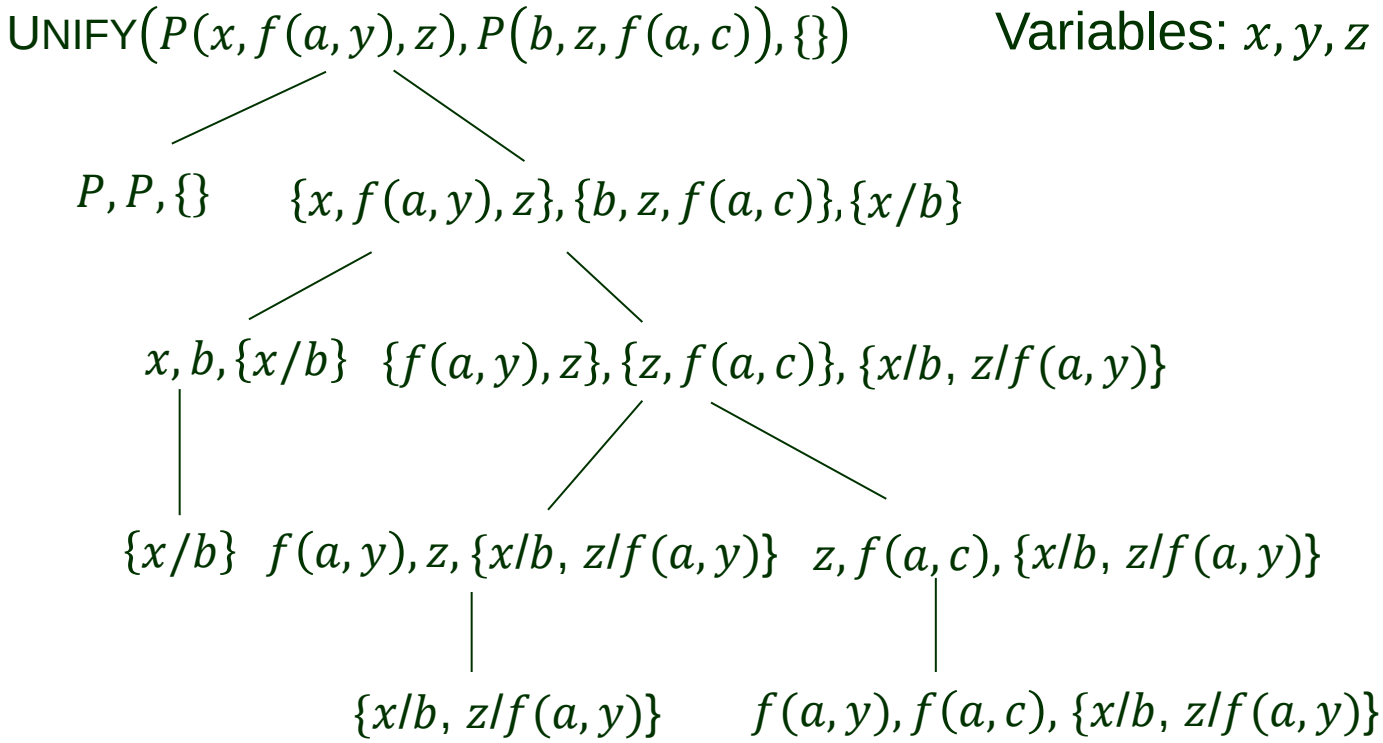
Unification Example



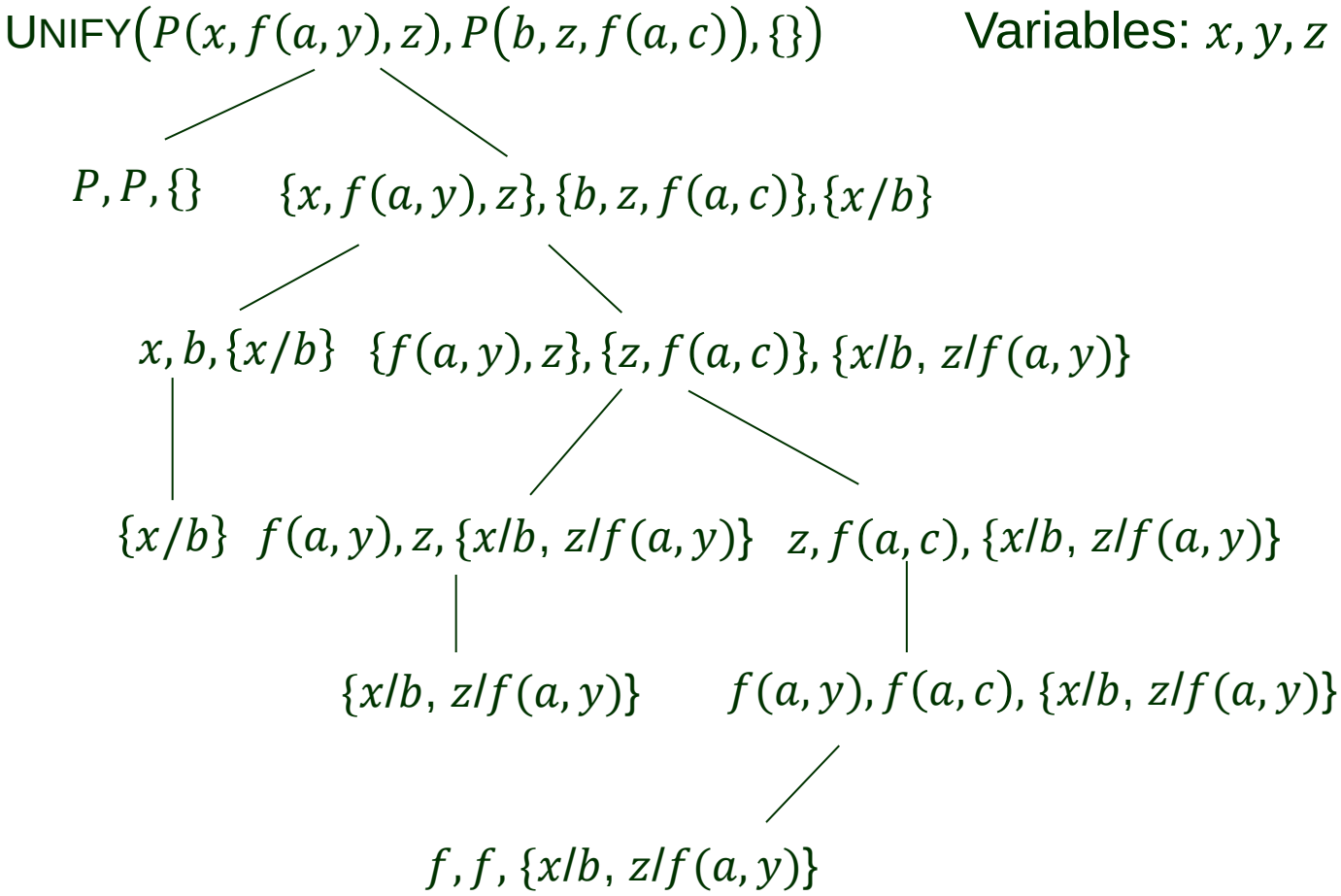
Unification Example



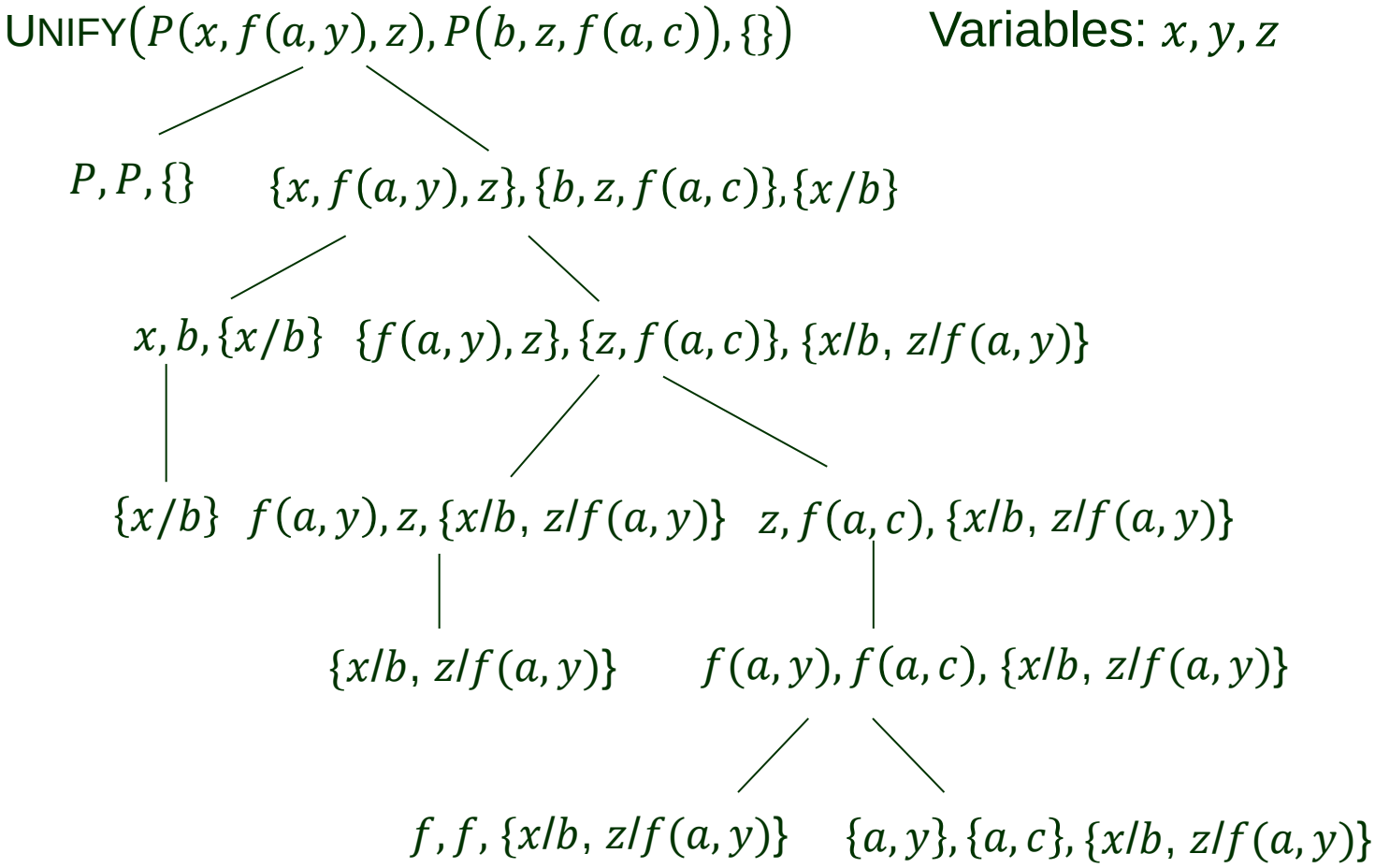
Unification Example



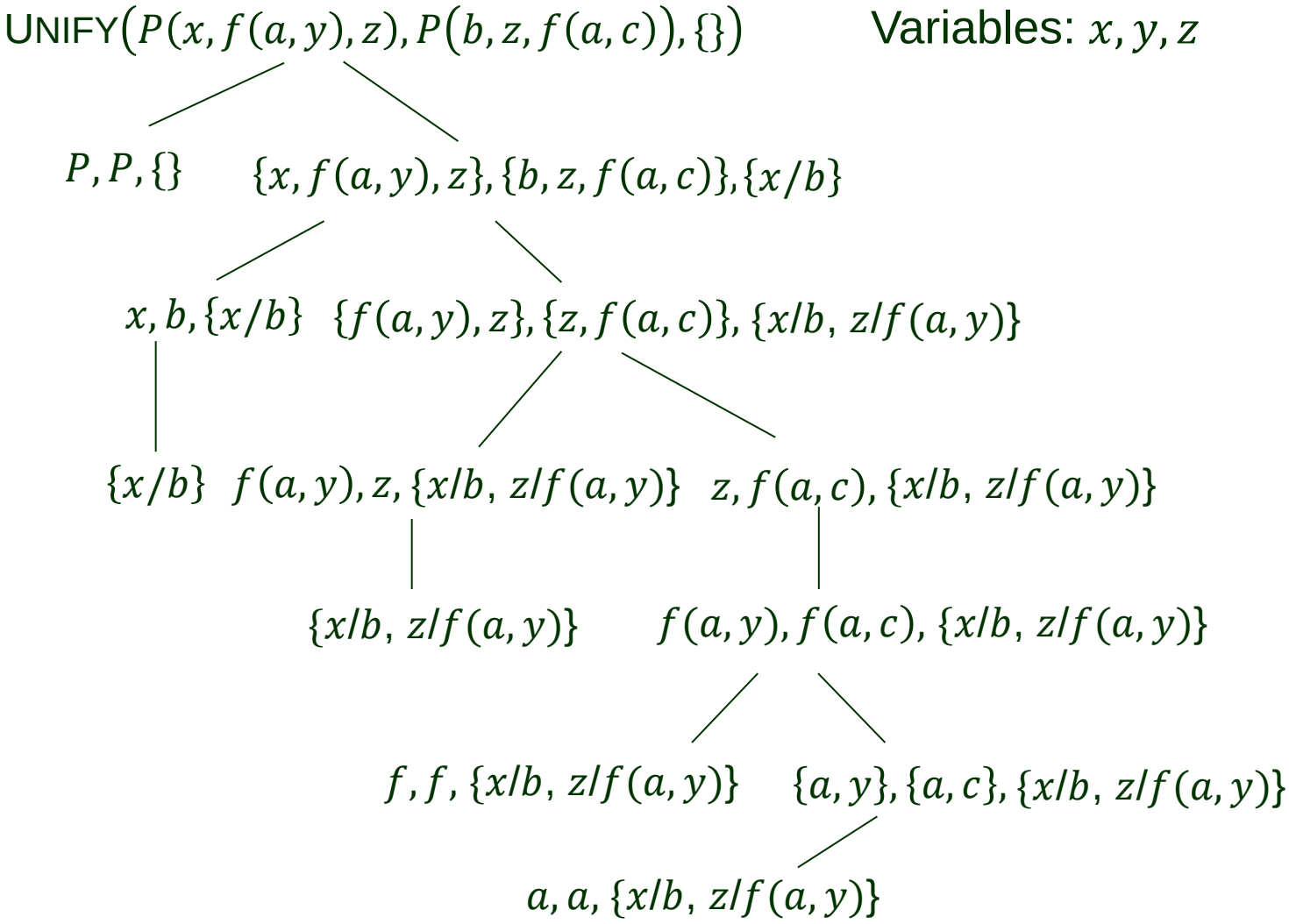
Unification Example



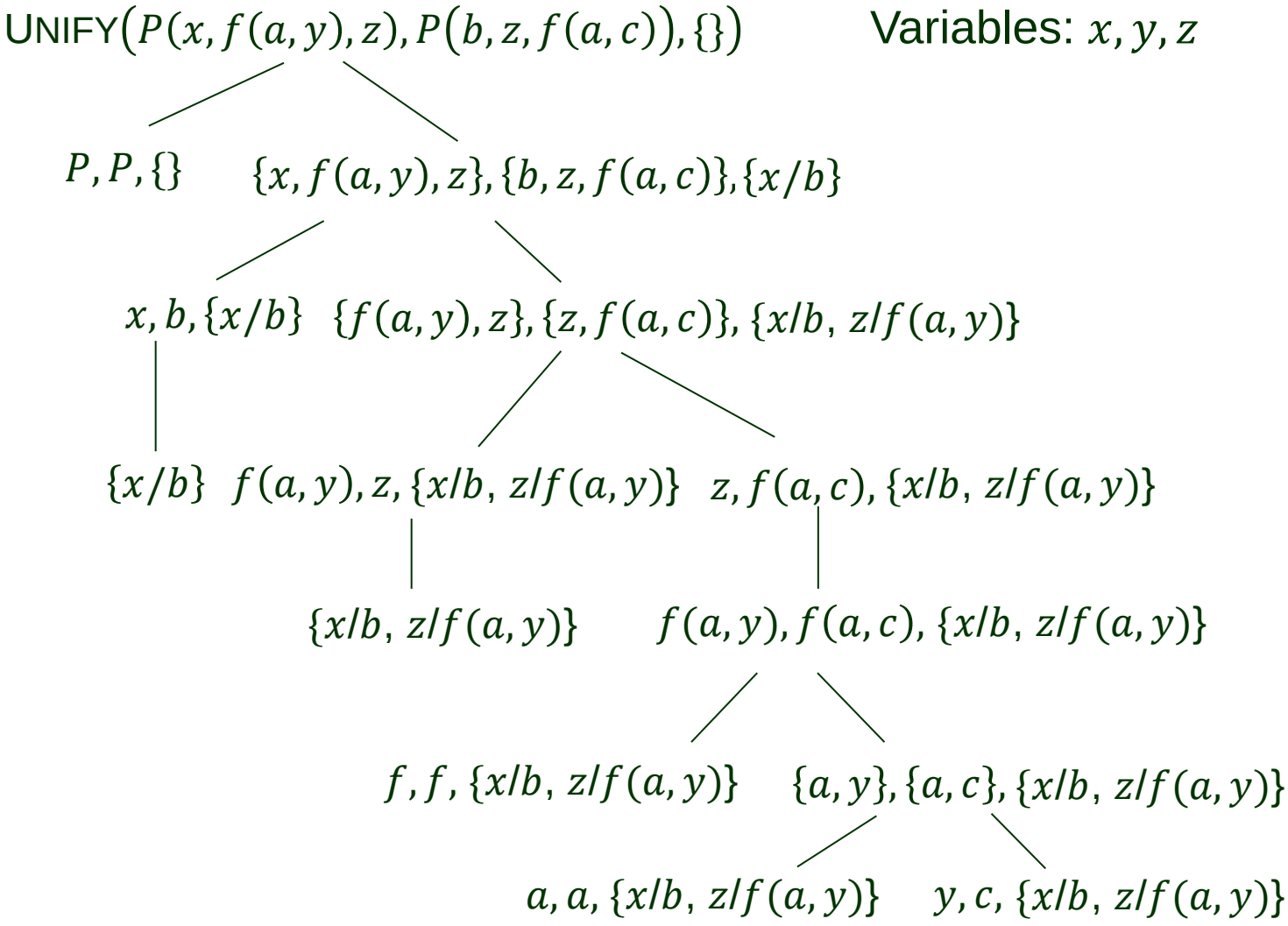
Unification Example



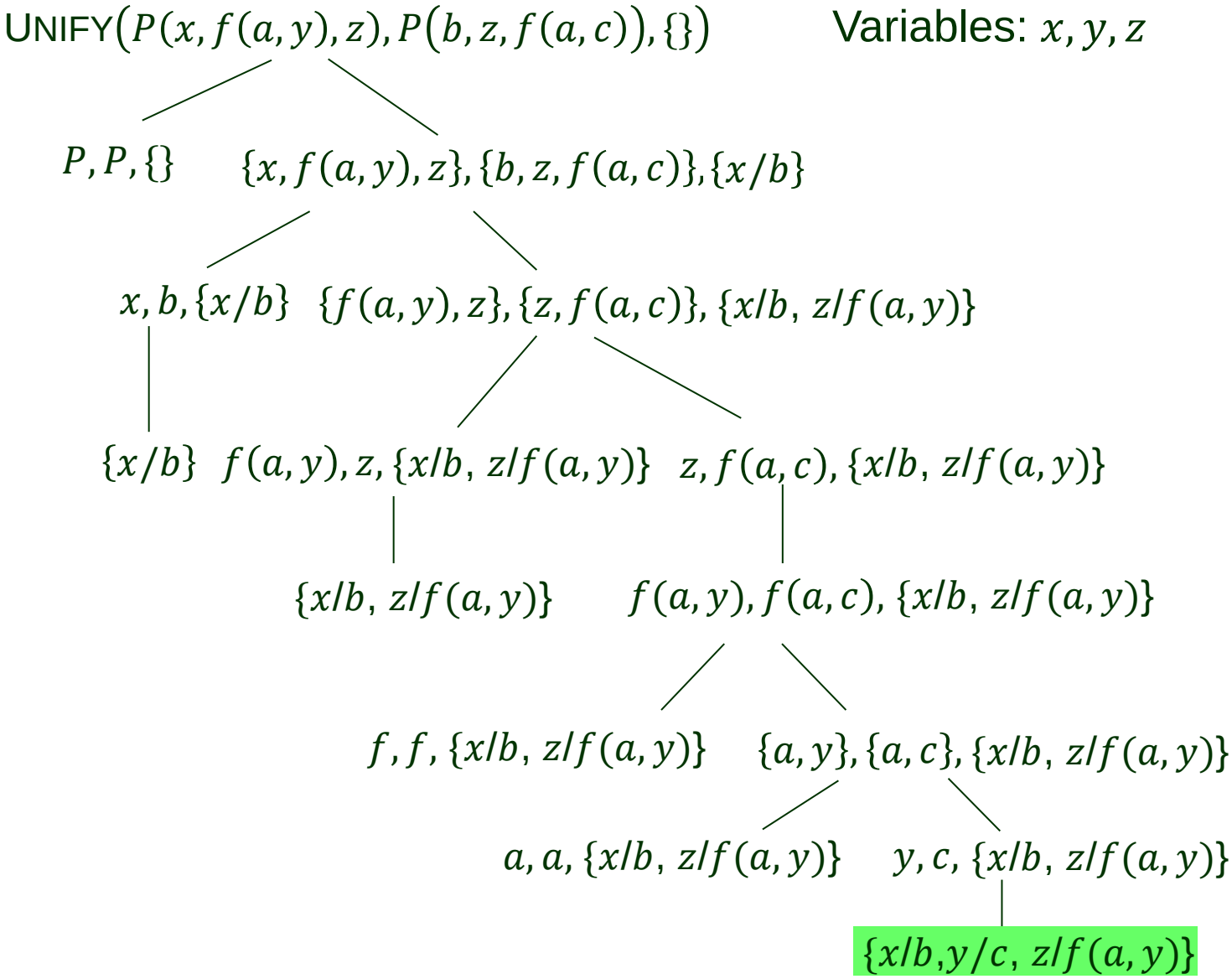
Unification Example



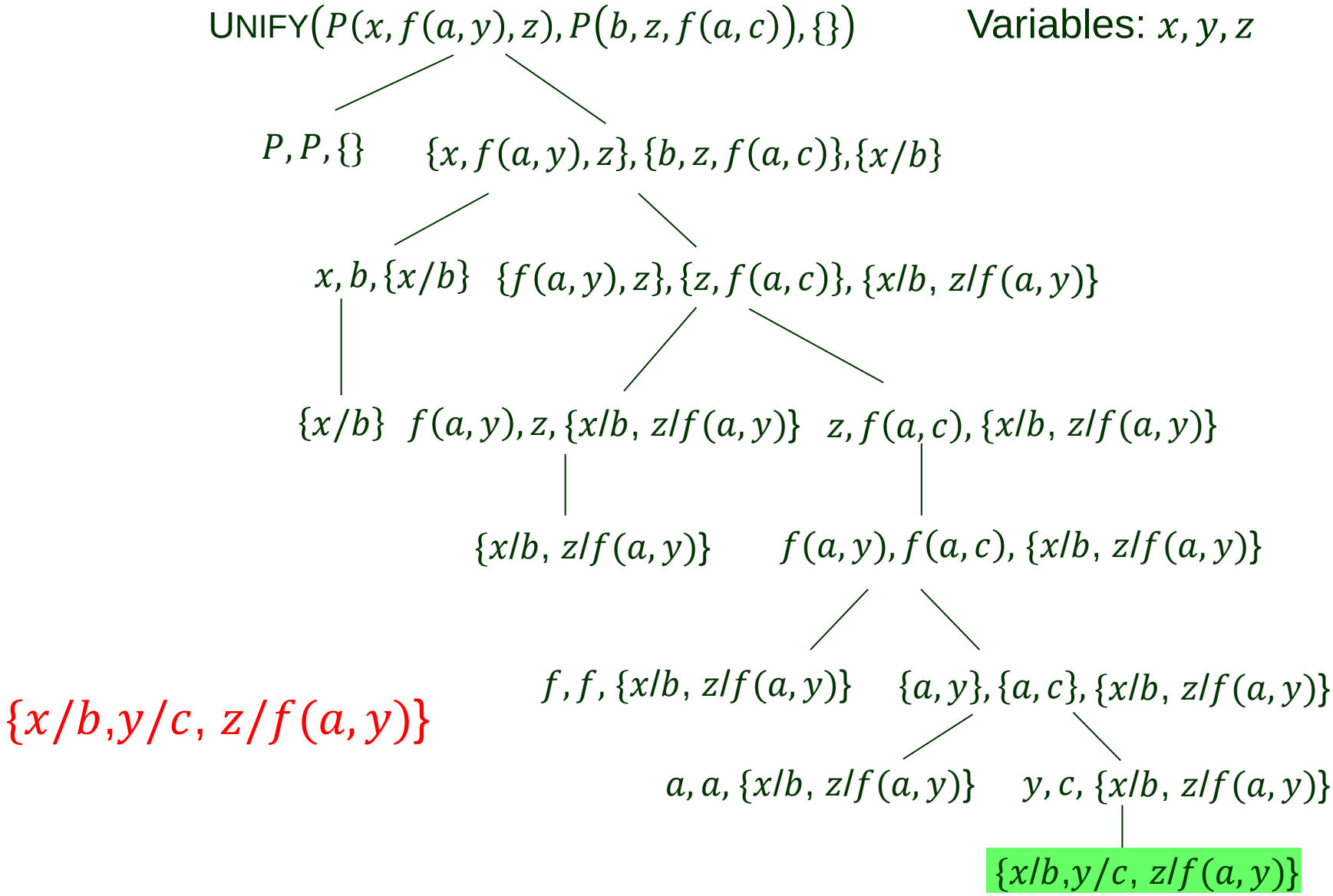
Unification Example



Unification Example



Unification Example



II. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which *exactly one* is positive (same definition as in propositional logic except now variables are allowed).

II. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which *exactly one* is positive (same definition as in propositional logic except now variables are allowed).

- single positive literal (i.e., fact in propositional logic if no variable)

$Bird(Ostrich)$

II. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which *exactly one* is positive (same definition as in propositional logic except now variables are allowed).

- single positive literal (i.e., fact in propositional logic if no variable)

$$Bird(Ostrich)$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal (i.e., definite clause in PL)

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$

II. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which *exactly one* is positive (same definition as in propositional logic except now variables are allowed).

- single positive literal (i.e., fact in propositional logic if no variable)

$$Bird(Ostrich)$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal (i.e., definite clause in PL)

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$

- variables allowed and implicitly *under universal quantification*

$$Human(x) \Rightarrow Fallible(x)$$

// interpreted as $\forall x \; Human(x) \Rightarrow Fallible(x)$

II. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which *exactly one* is positive (same definition as in propositional logic except now variables are allowed).

- single positive literal (i.e., fact in propositional logic if no variable)

$$Bird(Ostrich)$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal (i.e., definite clause in PL)

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$

- variables allowed and implicitly *under universal quantification*

$$Human(x) \Rightarrow Fallible(x)$$ // interpreted as $\forall x \;\; Human(x) \Rightarrow Fallible(x)$

$$Gate(g) \wedge Terminal(t) \Rightarrow g \neq t$$ // interpreted as $\forall g, t \;\; Gate(g) \wedge Terminal(t) \Rightarrow g \neq t$

II. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which *exactly one* is positive (same definition as in propositional logic except now variables are allowed).

- single positive literal (i.e., fact in propositional logic if no variable)

$$Bird(Ostrich)$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal (i.e., definite clause in PL)

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$

- variables allowed and implicitly *under universal quantification*

$$Human(x) \Rightarrow Fallible(x)$$ // interpreted as $\forall x \;\; Human(x) \Rightarrow Fallible(x)$

$$Gate(g) \wedge Terminal(t) \Rightarrow g \neq t$$ // interpreted as $\forall g, t \;\; Gate(g) \wedge Terminal(t) \Rightarrow g \neq t$

- existential quantifiers *not* allowed

Translation of Sentences

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.

Translation of Sentences

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.

"… *it is a crime for an American to sell weapons to hostile nations":*

Translation of Sentences

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.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

Translation of Sentences

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.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

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

Translation of Sentences

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.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

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

"*Nono … has some missles":*

Translation of Sentences

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.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

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

"*Nono … has some missles":* // $\exists x$ $Owns(Nono, x) \wedge Missile(x)$

Translation of Sentences

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.

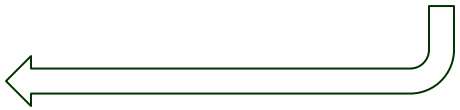
"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

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

"*Nono … has some missles":* // $\exists x$ $Owns(Nono, x) \wedge Missile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$



introducing a Skolem constant to eliminate ∃

Translation of Sentences

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.

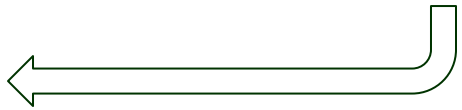
"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

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

"*Nono … has some missles":* // $\exists x$ $Owns(Nono, x) \wedge Missile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$



introducing a Skolem constant to eliminate ∃

"*All of its missiles were sold by Colonel West":* // $\forall x$ ...

Translation of Sentences

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.

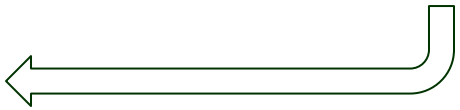
"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

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

"*Nono … has some missles":* // $\exists x \;\; Owns(Nono, x) \wedge Missile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$



introducing a Skolem constant to eliminate ∃

"*All of its missiles were sold by Colonel West":* // $\forall x$ ...

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

Completing the KB

Completing the KB

Need to know that missiles are weapons.

$$\mathit{Missile}(x) \Rightarrow \mathit{Weapon}(x)$$

Completing the KB

Need to know that missiles are weapons.

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

Also need to know that an enemy of America counts as “hostile”.

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

Completing the KB

Need to know that missiles are weapons.

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

Also need to know that an enemy of America counts as “hostile”.

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

“*West, who is American* …

$$American(West)$$

Completing the KB

Need to know that missiles are weapons.

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

Also need to know that 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 …*”

$$Enemy(Nono, America)$$

Completing the KB

Need to know that missiles are weapons.

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

Also need to know that 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 …*"

$$Enemy(Nono, America)$$

The *KB* consists of first-order definite clauses with no function symbols.
It is called a *Datalog*.

III. Simple Forward Chaining

1. Start from the known facts.
2. Trigger all the rules whose premises are satisfied.
3. Add their conclusions to the known facts.
4. Repeat steps 2 and 3 until one of the following situations occurs:
    a. The query is answered.
    b. No new facts are added.

III. Simple Forward Chaining

1. Start from the known facts.
2. Trigger all the rules whose premises are satisfied.
3. Add their conclusions to the known facts.
4. Repeat steps 2 and 3 until one of the following situations occurs:
    a. The query is answered.
    b. No new facts are added.

A *new fact* is not a renaming of a known fact.

III. Simple Forward Chaining

1. Start from the known facts.
2. Trigger all the rules whose premises are satisfied.
3. Add their conclusions to the known facts.
4. Repeat steps 2 and 3 until one of the following situations occurs:
    a. The query is answered.
    b. No new facts are added.

A *new fact* is not a renaming of a known fact.

Likes(x,*IceCream*) is a renaming of *Likes*(y,*IceCream*).
Both have the meaning: “Everyone likes ice cream”.

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sells(West, M_1, Nono)$

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sells(West, M_1, Nono)$

$Weapon(M_1)$

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sells(West, M_1, Nono)$

$Weapon(M_1)$

$Hostile(Nono)$

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sells(West, M_1, Nono)$

$Weapon(M_1)$

$Hostile(Nono)$

Iteration 2 adds:

Execution of Forward Chaining

KB:

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

$Owns(Nono, M_1)$

$Missile(M_1)$

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

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

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

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sells(West, M_1, Nono)$

$Weapon(M_1)$

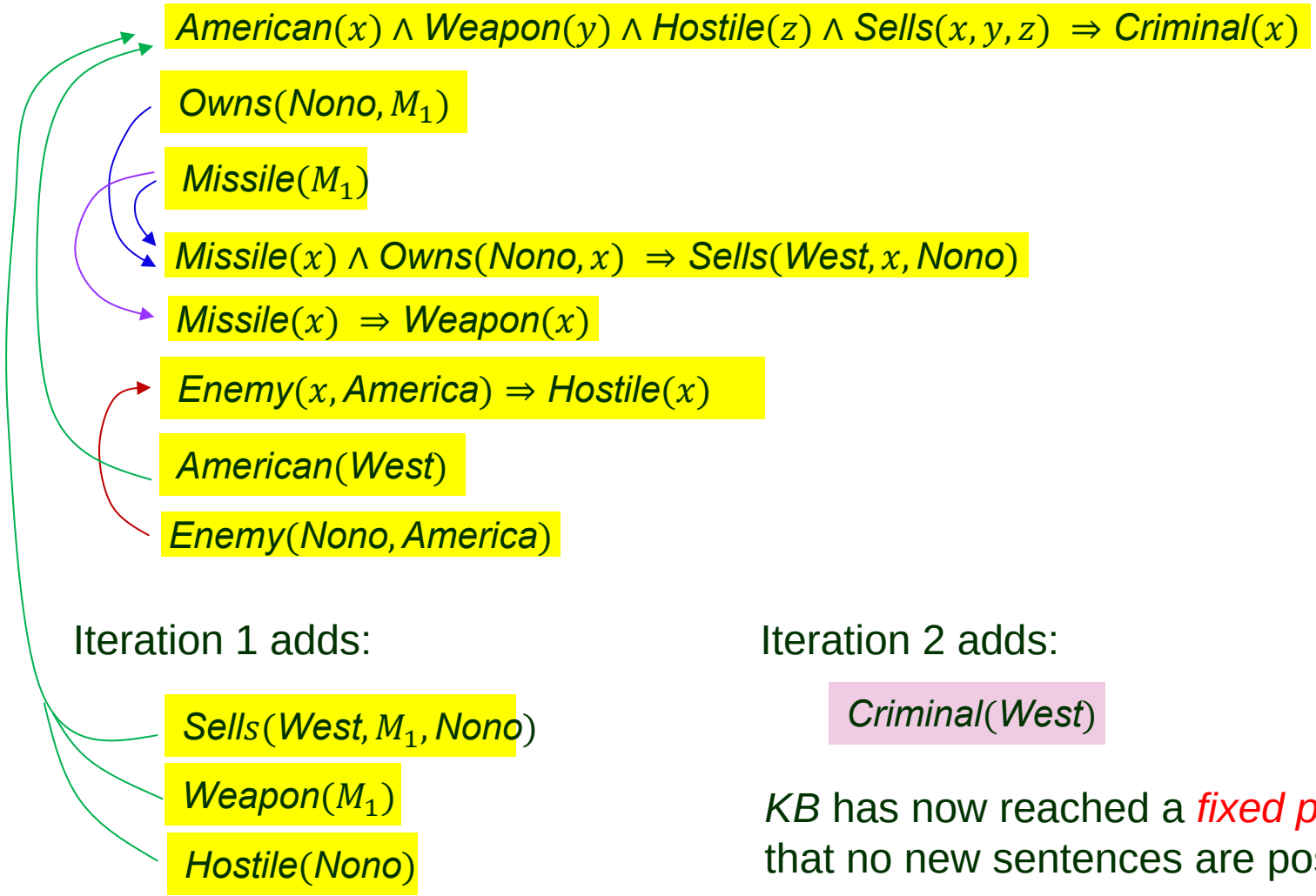
$Hostile(Nono)$

Iteration 2 adds:

$Criminal(West)$

Execution of Forward Chaining

KB:



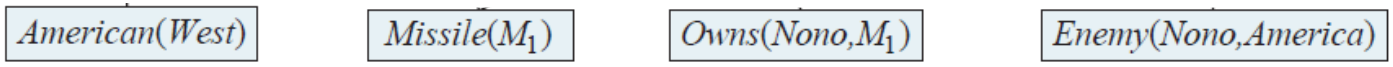
Iteration 1 adds:

Iteration 2 adds:

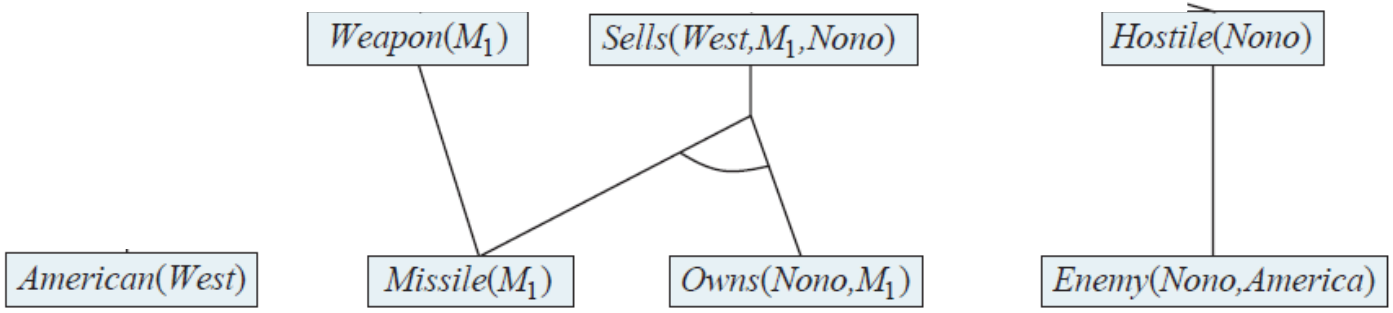
Criminal(*West*)

KB has now reached a *fixed point*, meaning that no new sentences are possible.

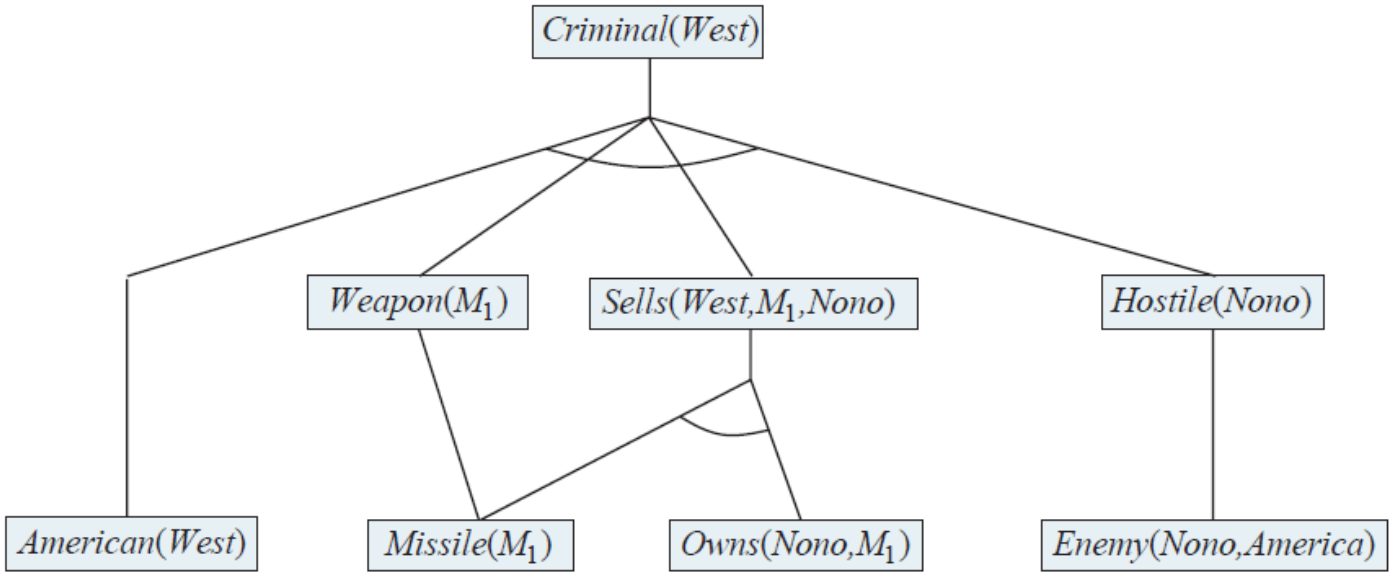
Proof Tree



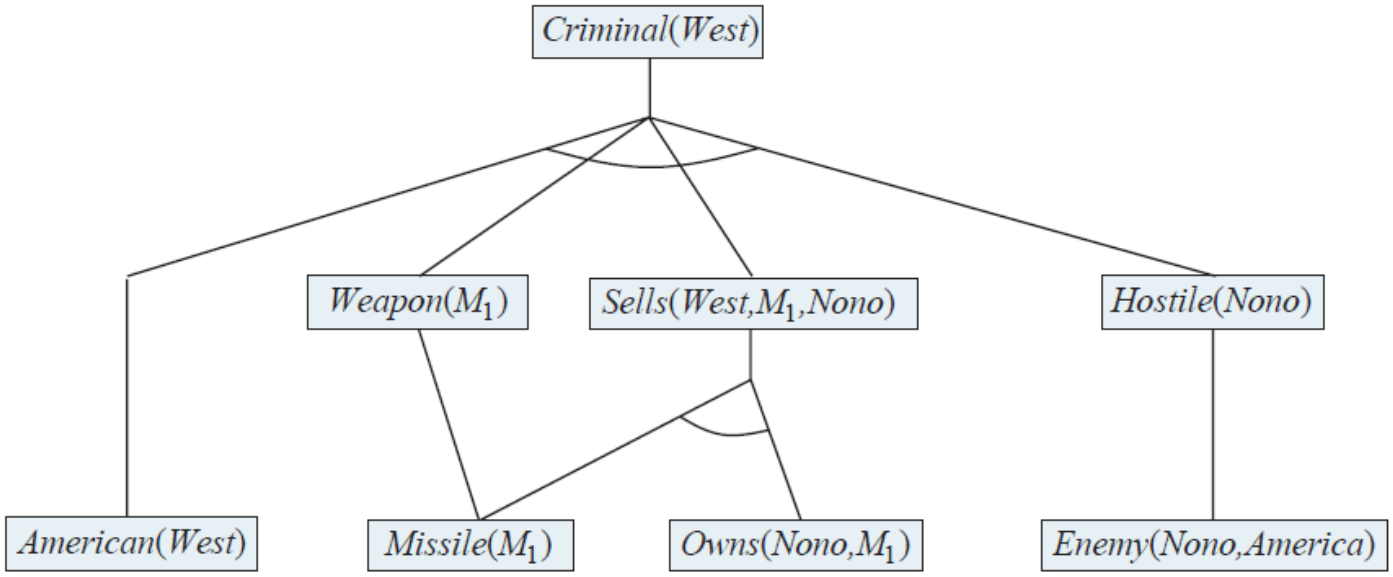
Proof Tree


Weapon(M_1)
Sells(West,M_1,Nono)
Hostile(Nono)
American(West)
Missile(M_1)
Owns(Nono,M_1)
Enemy(Nono,America)

Proof Tree


Criminal(West)
Weapon(M_1)
Sells(West,M_1,Nono)
Hostile(Nono)
American(West)
Missile(M_1)
Owns(Nono,M_1)
Enemy(Nono,America)

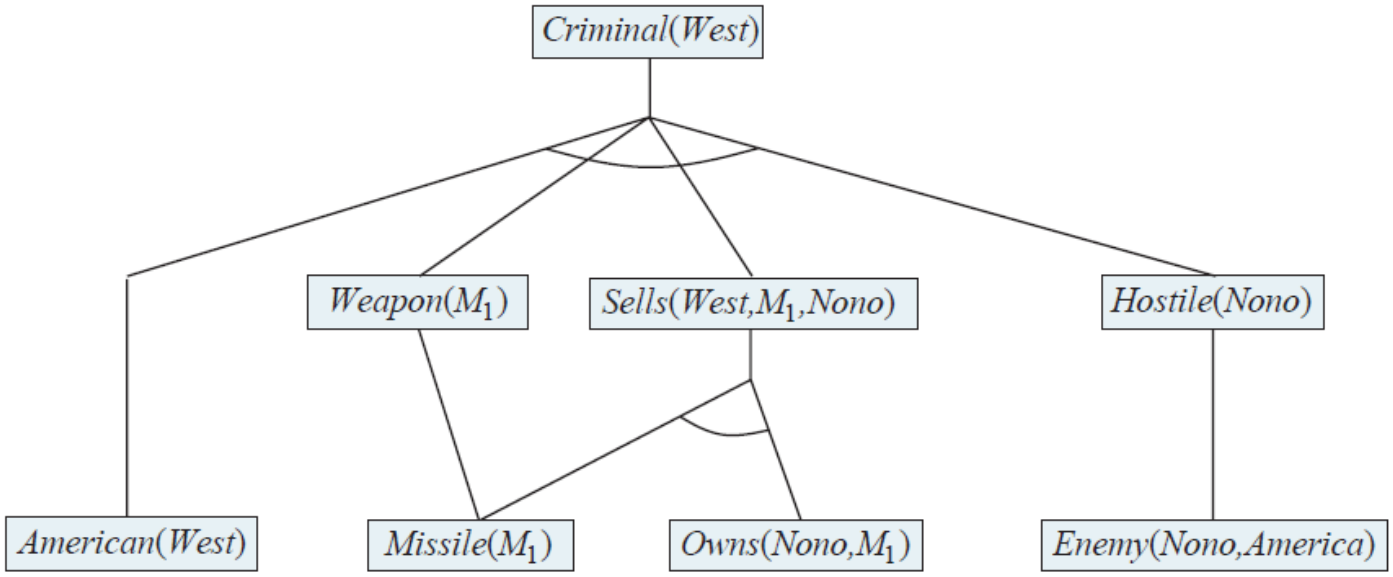
Proof Tree



- Soundness of forward chaining

  Every inference is an application of Generalized Modus Ponens.

Proof Tree



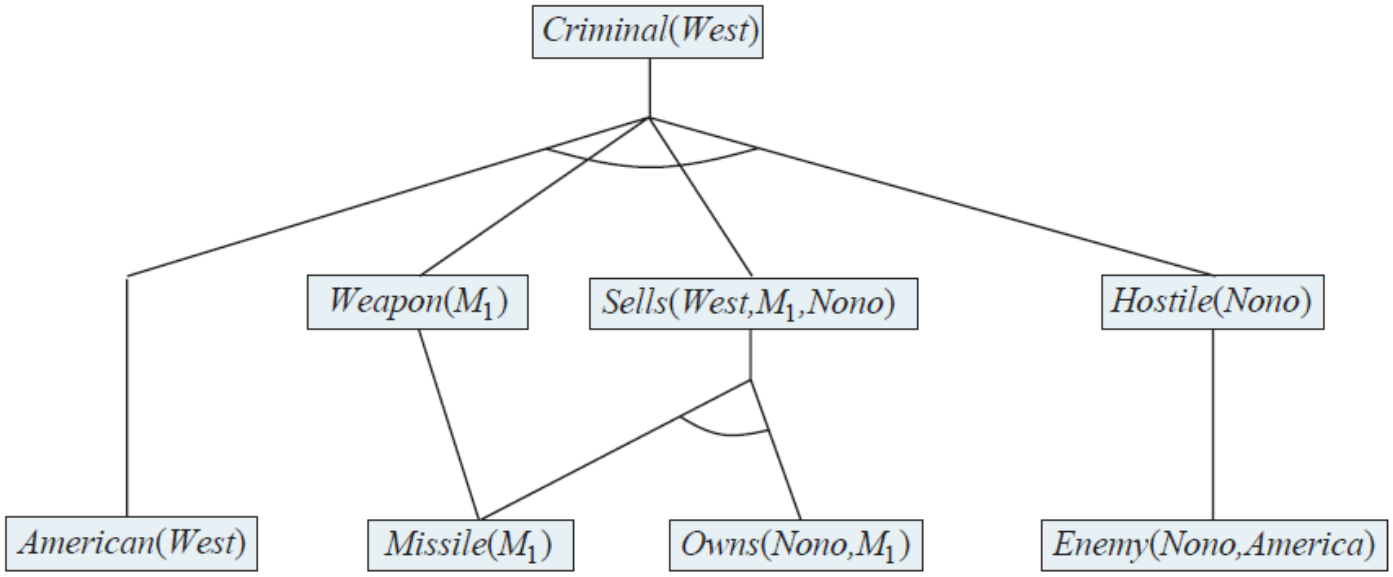
- Soundness of forward chaining

  Every inference is an application of Generalized Modus Ponens.

- Completeness

  - Easy to establish if no function symbols appears in the KB.

Proof Tree



- Soundness of forward chaining

  Every inference is an application of Generalized Modus Ponens.

- Completeness

  - Easy to establish if no function symbols appears in the KB.
  - Guaranteed except for a query with no answer, the algorithm could fail to terminate. E.g., $NatNum(0)$ and $NatNum(n) \Rightarrow NatNum(S(n))$.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⇨

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

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

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

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

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹ ♠ Exhaustively matches every rule against every fact.

♠ Rechecks every rule on each iteration (even with very few additions to *KB*).

♠ Generates many facts that are irrelevant to the goal.

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

♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:

♣ Find all the objects owned by Nono and, for each, check if it is a missile.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹
- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

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

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⇨

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

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

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono. More efficient!

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹
- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

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

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono. More efficient!
- ♦ How to order the conjuncts of the rule premise so they can be solved with the minimum total cost?

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

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

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono. More efficient!
- ♦ How to order the conjuncts of the rule premise so they can be solved with the minimum total cost?

NP-hard!

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

⟹ ♠ Exhaustively matches every rule against every fact.

♠ Rechecks every rule on each iteration (even with very few additions to *KB*).

♠ Generates many facts that are irrelevant to the goal.

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

♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:

♣ Find all the objects owned by Nono and, for each, check if it is a missile.

♣ Find all the missiles first and check if they are owned by Nono. More efficient!

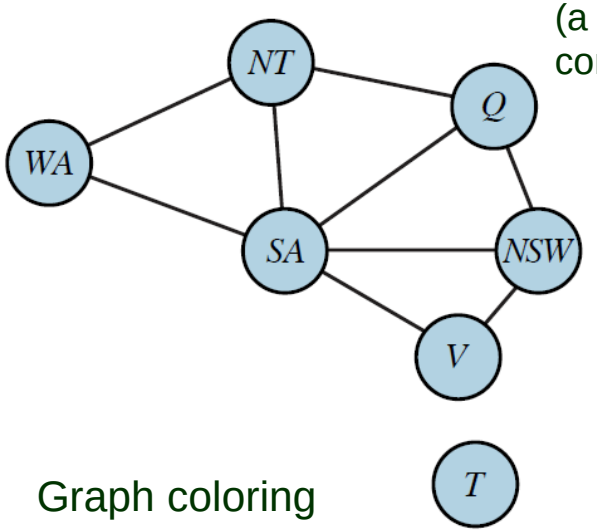
♦ How to order the conjuncts of the rule premise so they can be solved with the minimum total cost?

NP-hard!

Use a heuristic, e.g., the minimum-remaining-values (MRV) heuristic for CSPs.

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

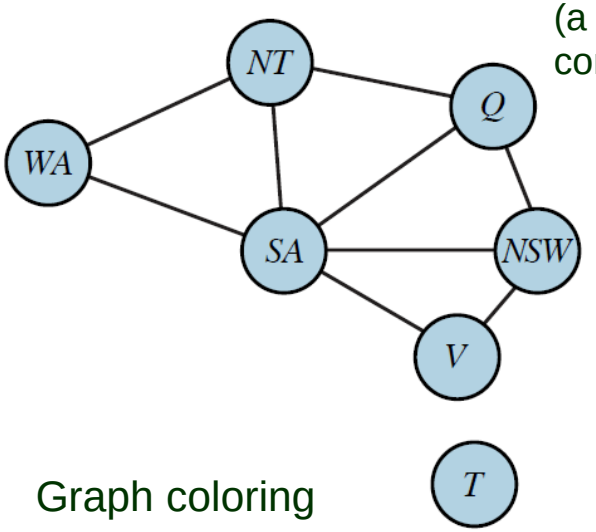
color of WA (a Skolem constant)

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*()

Diff(*Red*, *Blue*) *Diff*(*Red*, *Green*)
Diff(*Green*, *Red*) *Diff*(*Green*, *Blue*)
Diff(*Blue*, *Red*) *Diff*(*Blue*, *Green*)

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

color of WA (a Skolem constant)

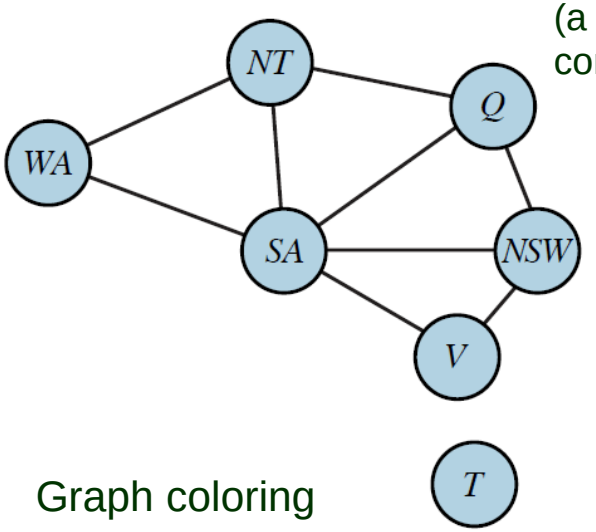
Diff(*wa*, *nt*) ∧ *Diff*(*wa*, *sa*) ∧
Diff(*nt*, *q*) ∧ *Diff*(*nt*, *sa*) ∧
Diff(*q*, *nsw*) ∧ *Diff*(*q*, *sa*) ∧
Diff(*nsw*, *v*) ∧ *Diff*(*nsw*, *sa*) ∧
Diff(*v*, *sa*) ⇒ *Colorable*()

Diff(*Red*, *Blue*) *Diff*(*Red*, *Green*)
Diff(*Green*, *Red*) *Diff*(*Green*, *Blue*)
Diff(*Blue*, *Red*) *Diff*(*Blue*, *Green*)

Constraint satisfaction is NP-hard.

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

color of WA (a Skolem constant)

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*()

Diff(*Red*, *Blue*) *Diff*(*Red*, *Green*)
Diff(*Green*, *Red*) *Diff*(*Green*, *Blue*)
Diff(*Blue*, *Red*) *Diff*(*Blue*, *Green*)

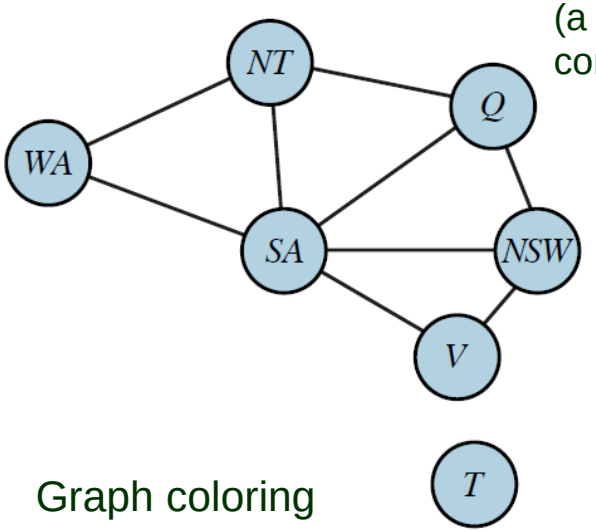
Constraint satisfaction is NP-hard.

⇩

Matching a definite clause against a set of facts (under unification) is NP-hard.

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

color of WA (a Skolem constant)

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*()

Diff(*Red, Blue*) *Diff*(*Red, Green*)
Diff(*Green, Red*) *Diff*(*Green, Blue*)
Diff(*Blue, Red*) *Diff*(*Blue, Green*)

Constraint satisfaction is NP-hard.

⇩

Matching a definite clause against a set of facts (under unification) is NP-hard.

Good news View every Datalog clause as a CSP and apply heuristics for CSPs (e.g., tree structure, cutset conditioning, etc.).

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i - 1$.
- Only a small fraction of the rules are triggered by a fact.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i - 1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i - 1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i - 1$.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i - 1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i - 1$.
2. Extends the substitution to match p_i with p_i'.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i - 1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i - 1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.
4. The remaining conjunts are matched with facts from iterations at or before $i-1$.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.
4. The remaining conjunts are matched with facts from iterations at or before $i-1$.

E.g., the Rete algorithm

IV. Backward Chaining

Works like AND/OR search:

- OR
  - The goal query can be proved by any rule in the *KB*.
  - A query containing a variable, e.g., $Person(x)$ can be proved in multiple ways.
- AND: all the conjuncts in the premise of a clause must be proved.

IV. Backward Chaining

Works like AND/OR search:

- OR
  - The goal query can be proved by any rule in the *KB*.
  - A query containing a variable, e.g., $Person(x)$ can be proved in multiple ways.
- AND: all the conjuncts in the premise of a clause must be proved.

How does it work?

IV. Backward Chaining

Works like AND/OR search:

- OR
  - The goal query can be proved by any rule in the *KB*.
  - A query containing a variable, e.g., $Person(x)$ can be proved in multiple ways.
- AND: all the conjuncts in the premise of a clause must be proved.

How does it work?

- Fetch all clauses that unify with the goal.
- Rename the variables in every such clause to be brand-new.
- Prove every conjunct in the clause by keeping track of the expanded substitution as it goes.

Depth-First Proof Tree

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

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

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

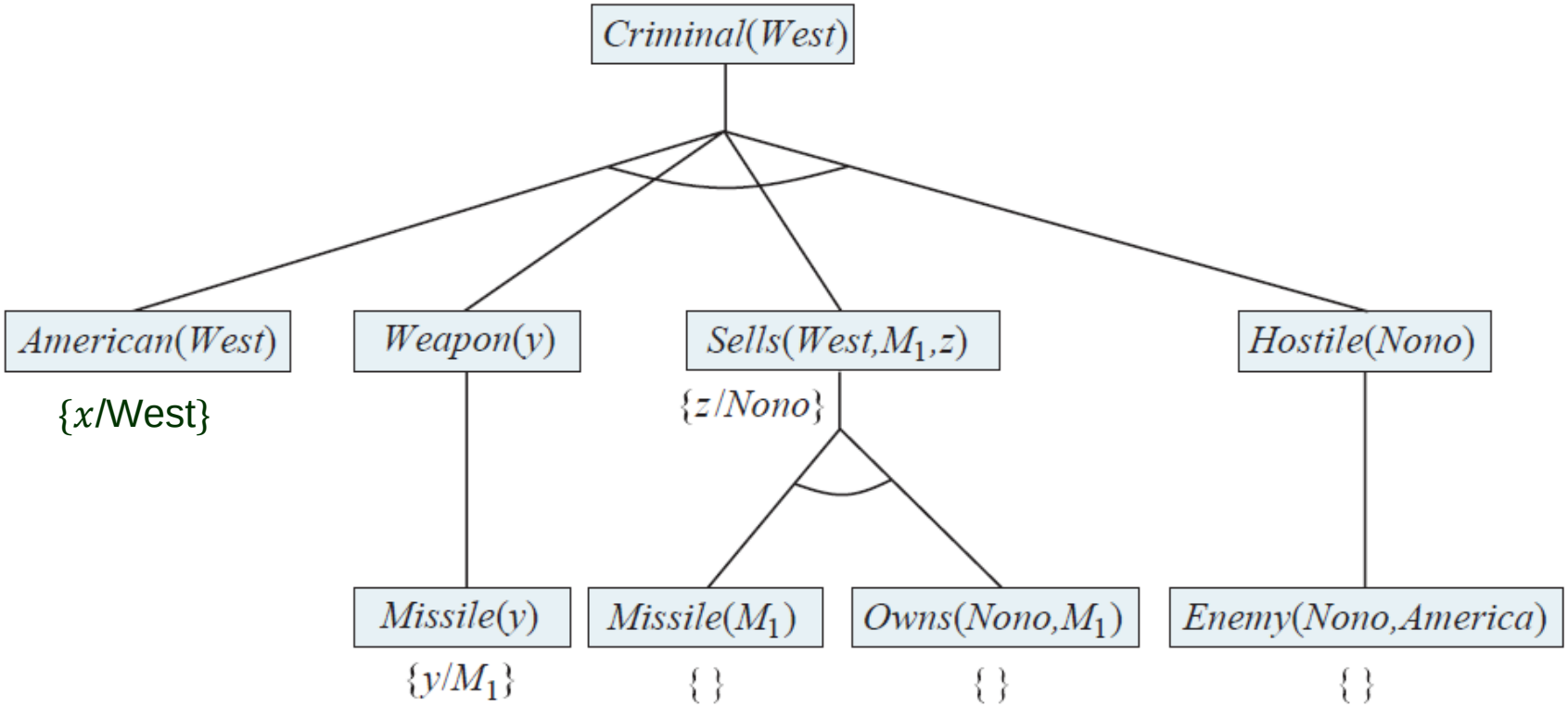
$Enemy(x, America) \Rightarrow Hostile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$American(West)$

$Enemy(Nono, America)$



Depth-First Proof Tree

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

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

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

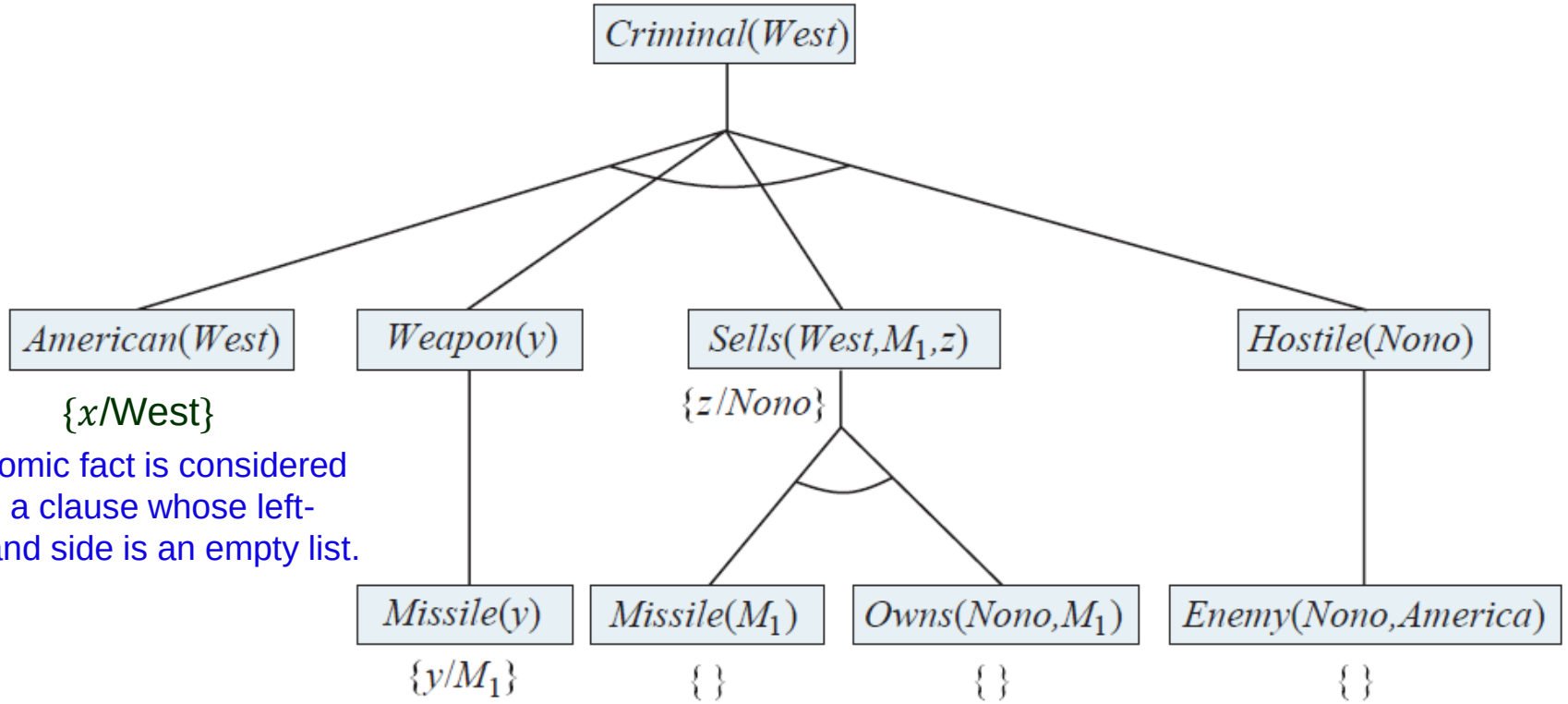
$Enemy(x, America) \Rightarrow Hostile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$American(West)$

$Enemy(Nono, America)$



Atomic fact is considered as a clause whose left-hand side is an empty list.