Backward chaining

When a query q is asked
  if a matching fact q' is known, return the unifier
  for each rule whose consequent q' matches q
    attempt to prove each premise of the rule by backward chaining

(Some added complications in keeping track of the unifiers)

(More complications help to avoid infinite loops)

Two versions: find <u>any</u> solution, find <u>all</u> solutions

Backward chaining is the basis for <u>logic programming</u>, e.g., Prolog

A simple example

- B^C=> G
- A^G=> I
- D^G=>J
- E=>   C
- D^C=>K
- F=>C

- Q: I?

A simple example

- B^C=> G
- A^G=> I
- D^G=>J
- E=>   C
- D^C=>K
- F=>C
- Q: I?

1. A^G
2. A?
   1. USER
3. G?
   1. B^C
      1. USER
      2. E v F

Backward Chaining

- Current knowledge:
  - hurts(x, head)

- What implications can lead to this fact?
  - kicked(x, head)
  - fell_on(x, head)
  - brain_tumor(x)
  - hangover(x)

- What facts do we need in order to prove these?

Backward Chaining

- The algorithm (available in detail in AIMA text):
  - a knowledge base KB
  - a desired conclusion c or question q
  - finds all sentences that are answers to q in KB *or* proves c
    - if q is directly provable by premises in KB, infer q and remember how q was inferred (building a list of answers).
    - find all implications that have q as a consequent.
    - for each of these implications, find out whether all of its premises are now in the KB, in which case infer the consequent and add it to the KB, remembering how it was inferred. If necessary, attempt to prove the implication also via backward chaining
    - premises that are conjuncts are processed one conjunct at a time

Backward chaining algorithm

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $goals$, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution $\{\,\}$
  local variables: ans, a set of substitutions, initially empty

  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow \text{SUBST}(\theta, \text{FIRST}(goals))$
  for each r **in** KB where $\text{STANDARDIZE-APART}(r) = (\, p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow \text{UNIFY}(q, q')$ succeeds
   $ans \leftarrow \text{FOL-BC-ASK}(KB, [p_1, \ldots, p_n | \text{REST}(goals)], \text{COMPOSE}(\theta, \theta')) \cup ans$
  return ans

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p))$$

Backward Chaining

- Question: Has Reality Man done anything criminal?
  - Criminal(Reality Man)

- Possible answers:
  - Steal(x, y) $\Rightarrow$ Criminal(x)
  - Kill(x, y) $\Rightarrow$ Criminal(x)
  - Grow(x, y) $\wedge$ Illegal(y) $\Rightarrow$ Criminal(x)
  - HaveSillyName(x) $\Rightarrow$ Criminal(x)
  - Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$Criminal(x)

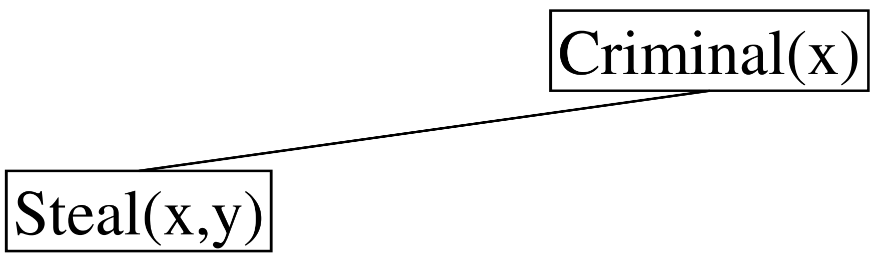
Backward Chaining

- Question: Has Reality Man done anything criminal?

$\boxed{\text{Criminal(x)}}$

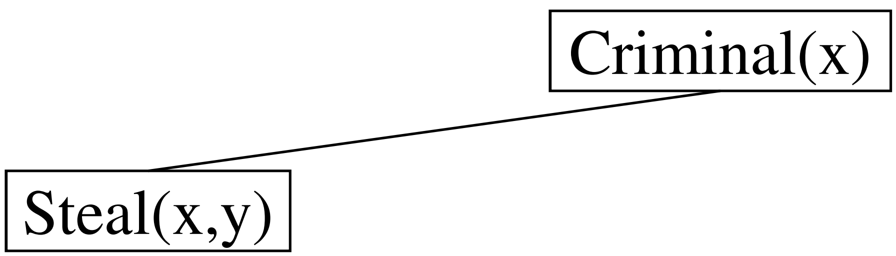
Backward Chaining

- Question: Has Reality Man done anything criminal?



Backward Chaining

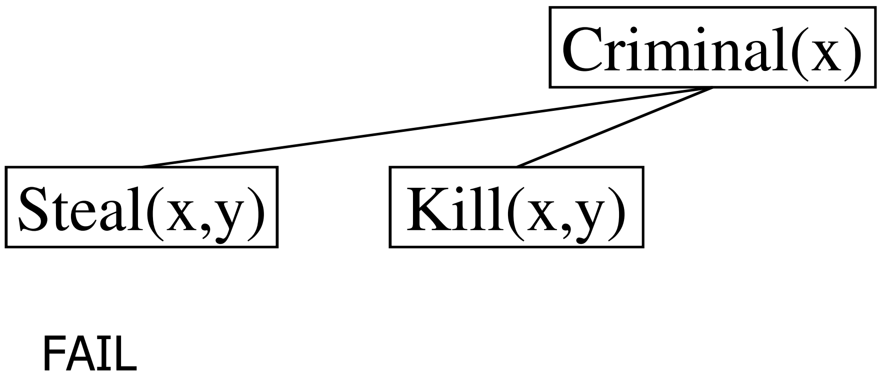
- Question: Has Reality Man done anything criminal?



FAIL

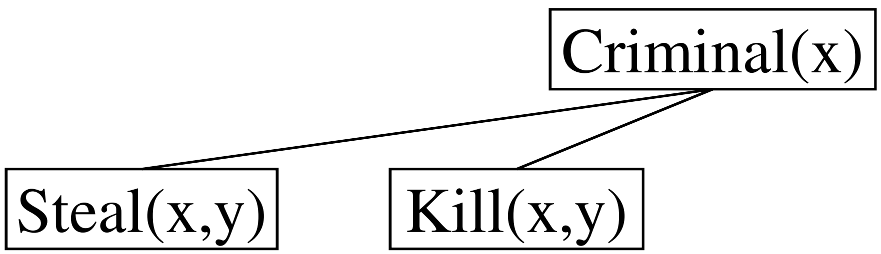
Backward Chaining

- Question: Has Reality Man done anything criminal?



Backward Chaining

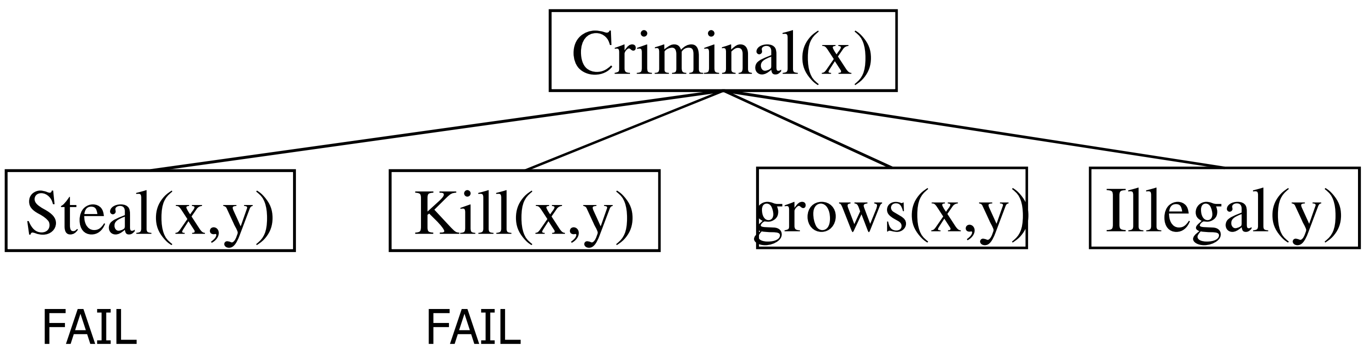
- Question: Has Reality Man done anything criminal?



FAIL FAIL

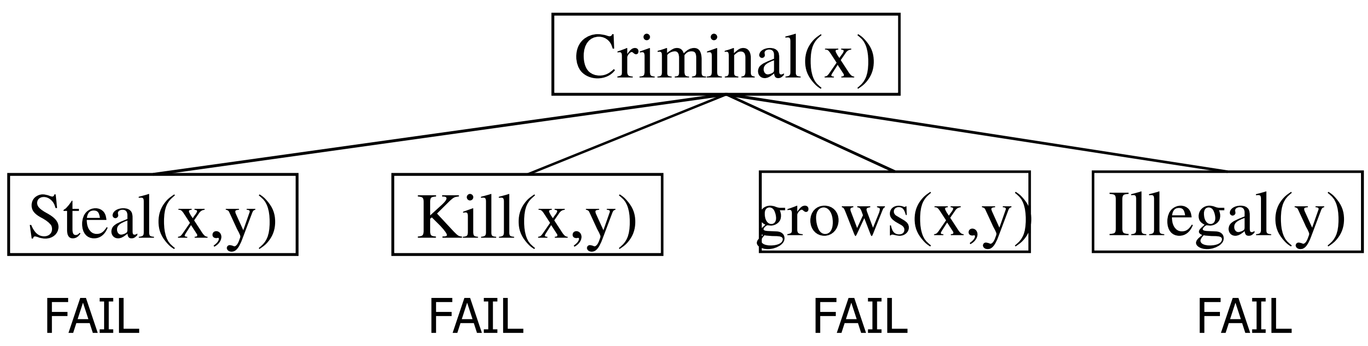
Backward Chaining

- Question: Has Reality Man done anything criminal?



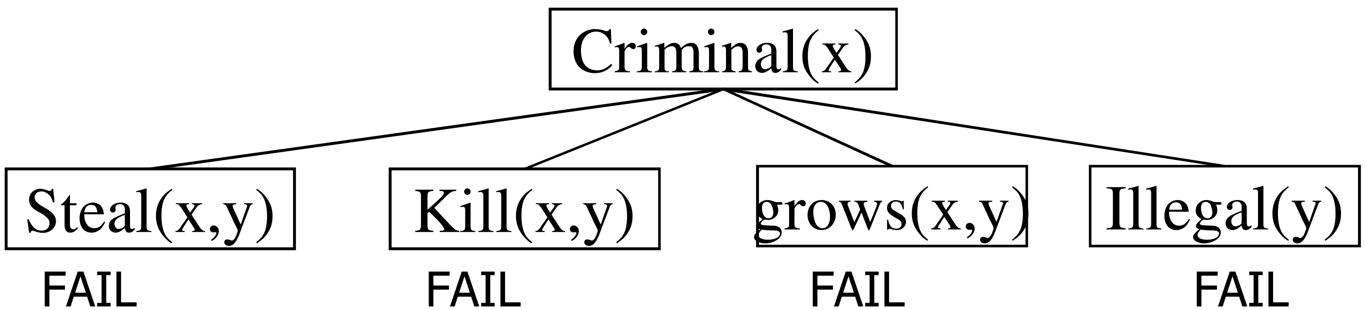
Backward Chaining

- Question: Has Reality Man done anything criminal?



Backward Chaining

- Question: Has Reality Man done anything criminal?



- Backward Chaining is a depth-first search: in any knowledge base of realistic size, many search paths will result in failure.

Backward Chaining

- Question: Has Reality Man done anything criminal?
- We will use the same knowledge as in our forward-chaining version of this example:

  Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$ Criminal(x)

  Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Provide(Reality Man, friends, x)

  Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Emulator(x)

  Programmer(Reality Man)

  People(friends)

  Software(U64)

  Use(friends, U64)

  Runs(U64, N64 games)

Backward Chaining

- Question: Has Reality Man done anything criminal?

$\boxed{\text{Criminal}(x)}$

Backward Chaining

- Question: Has Reality Man done anything criminal?

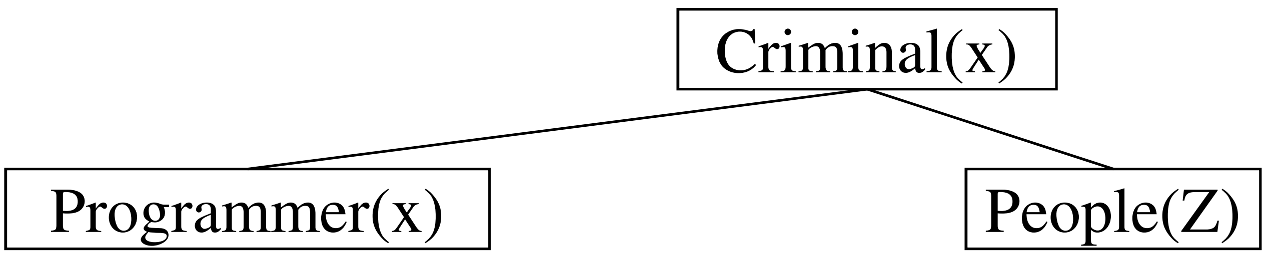
Criminal(x)

Programmer(x)

Yes, {x/Reality Man}

Backward Chaining

- Question: Has Reality Man done anything criminal?

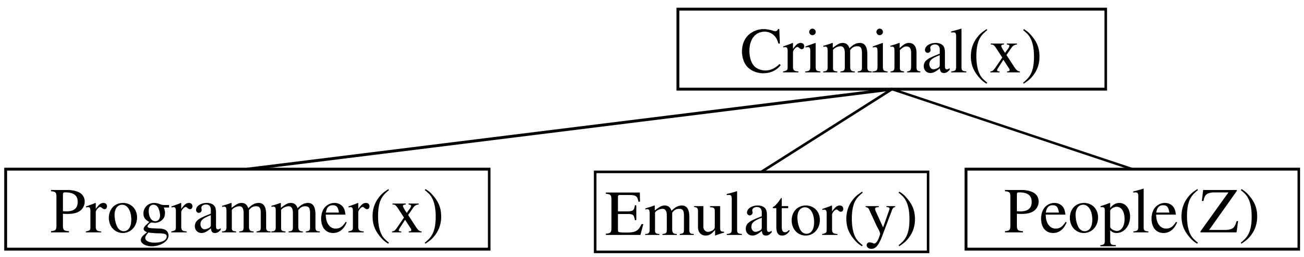


Yes, {x/Reality Man}

Yes, {z/friends}

Backward Chaining

- Question: Has Reality Man done anything criminal?

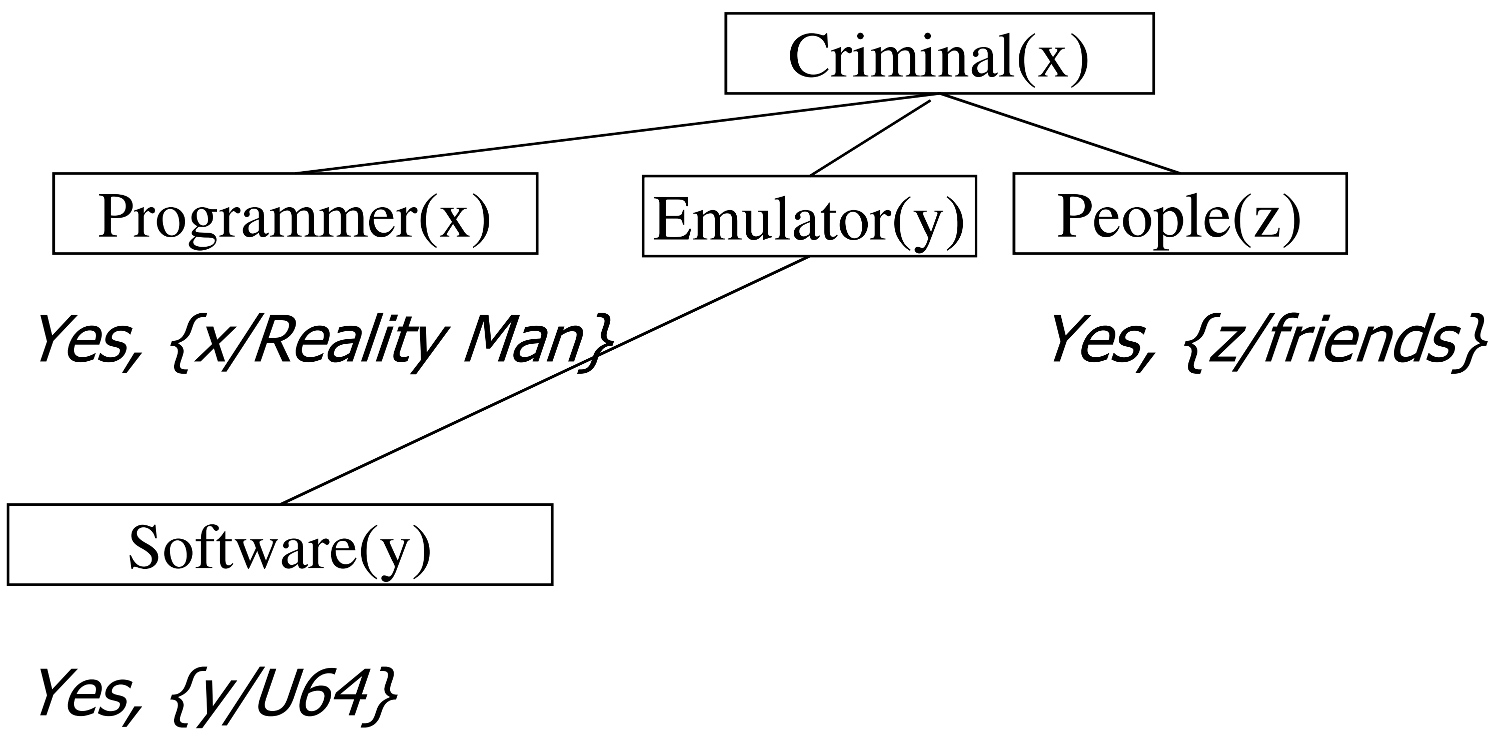


Yes, {x/Reality Man}

Yes, {z/friends}

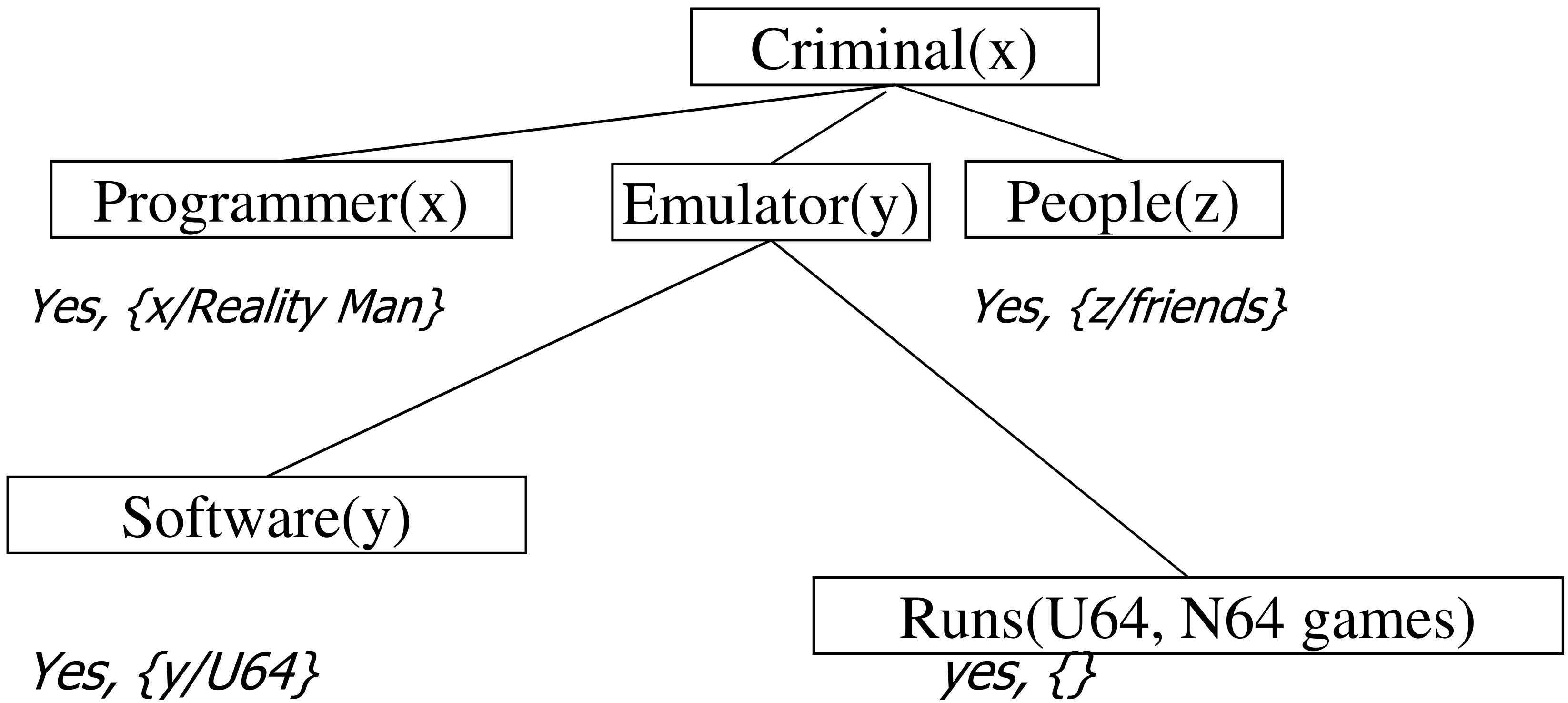
Backward Chaining

- Question: Has Reality Man done anything criminal?



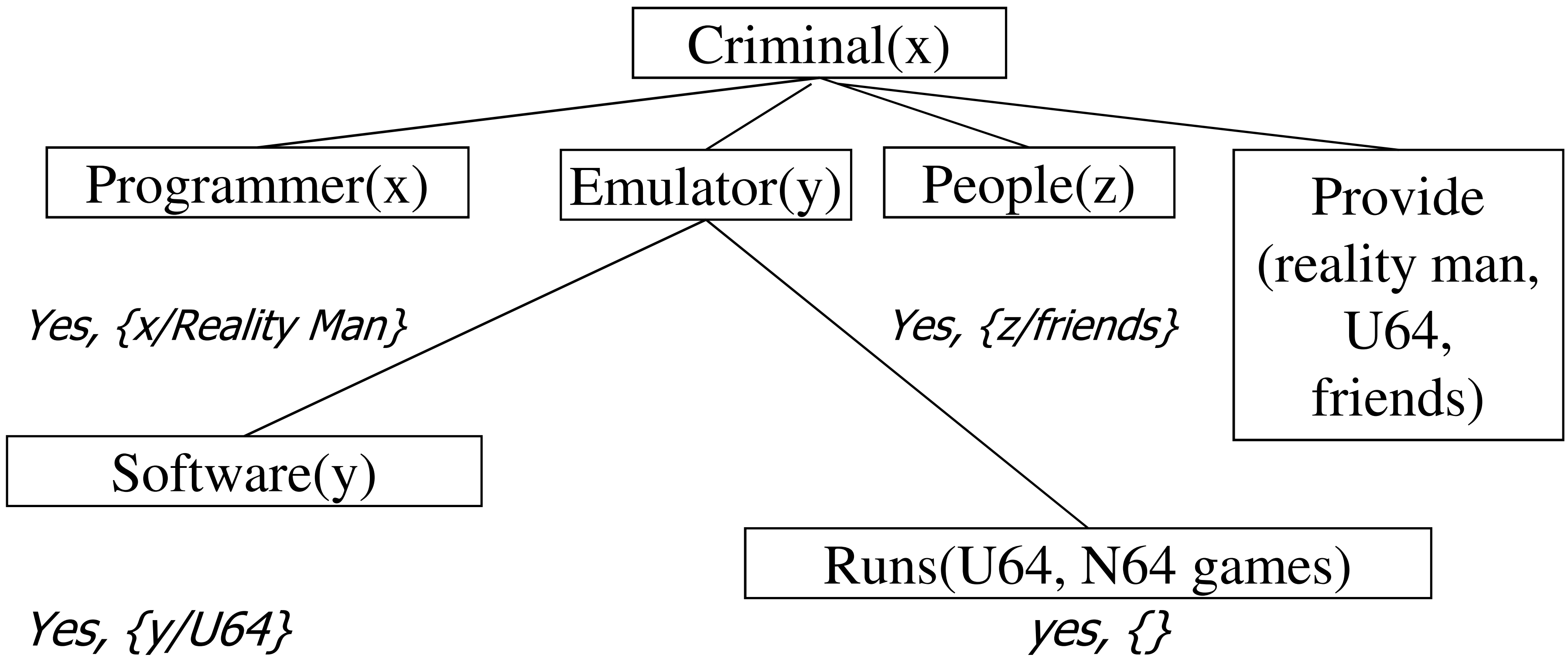
Backward Chaining

- Question: Has Reality Man done anything criminal?



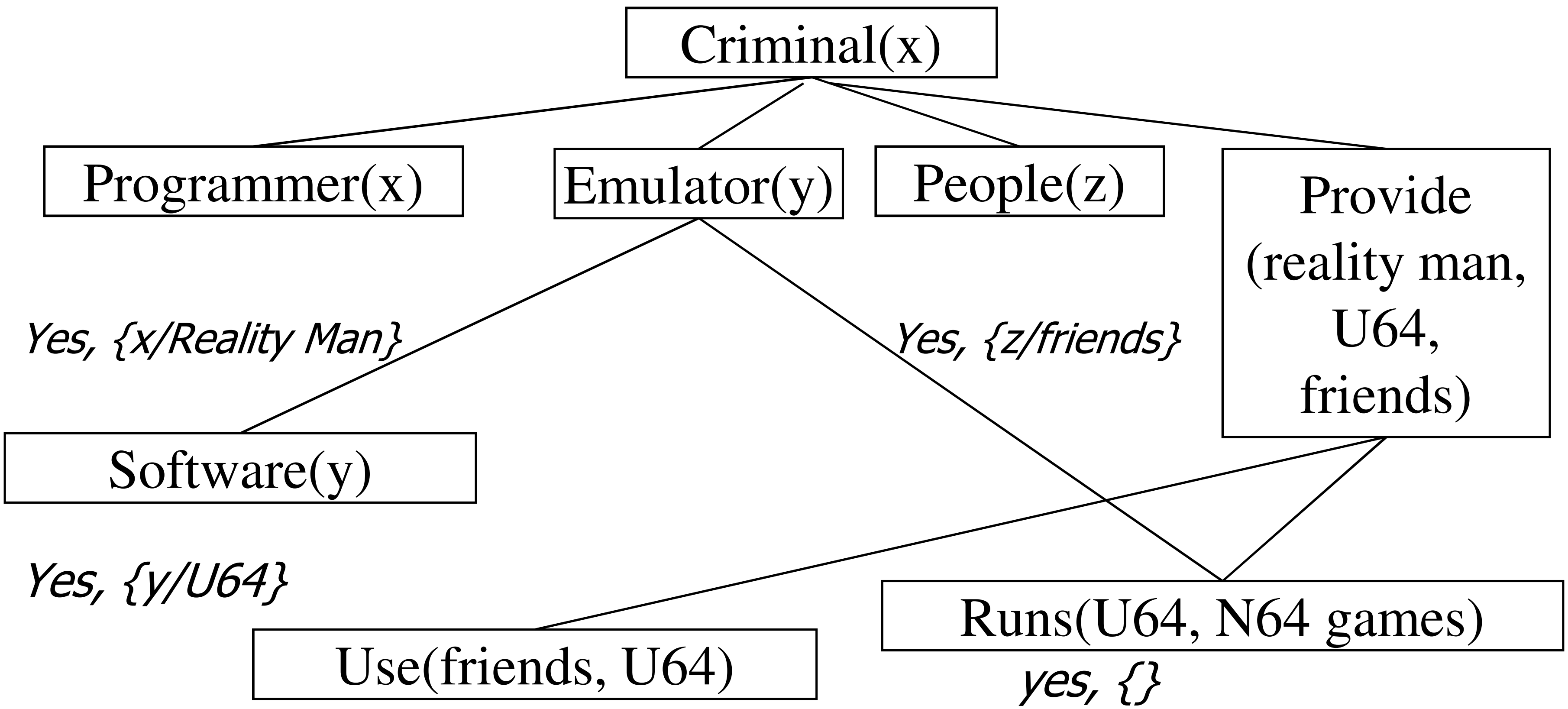
Backward Chaining

- Question: Has Reality Man done anything criminal?



Backward Chaining

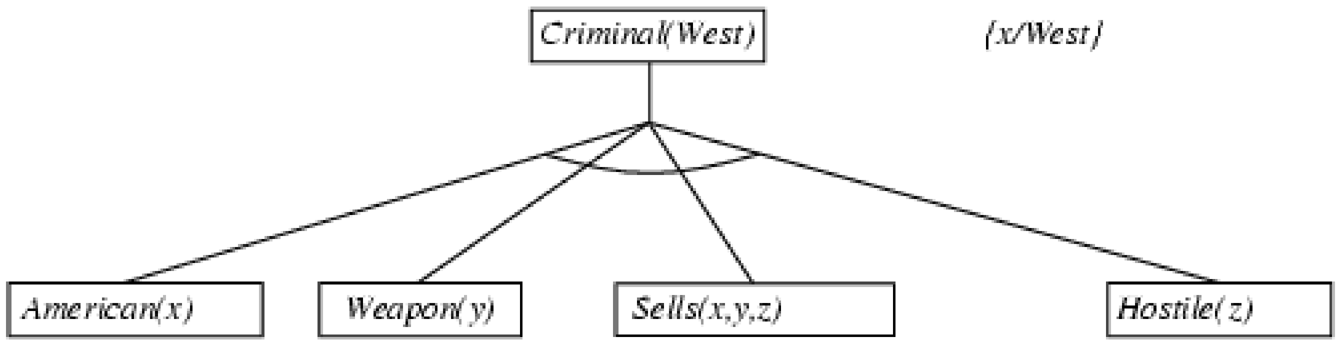
- Question: Has Reality Man done anything criminal?



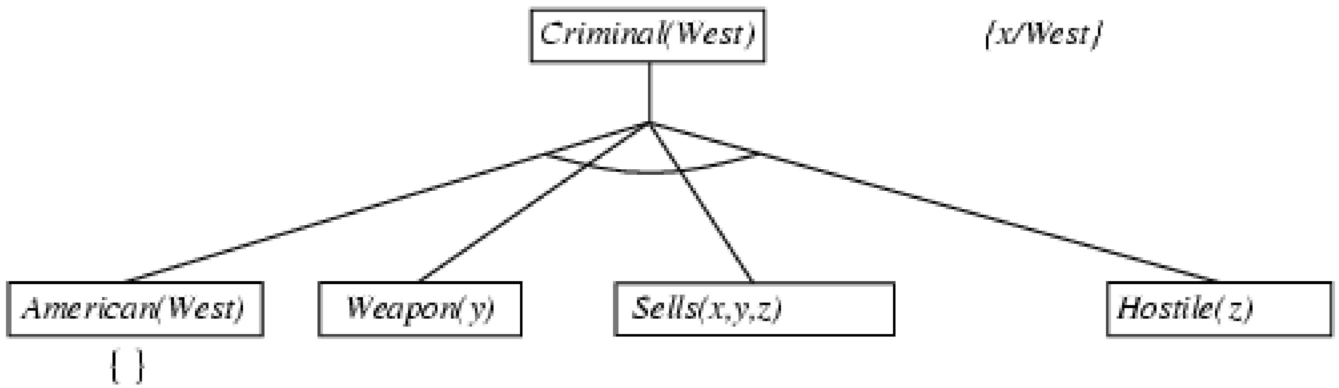
Backward chaining example

$\boxed{Criminal(West)}$

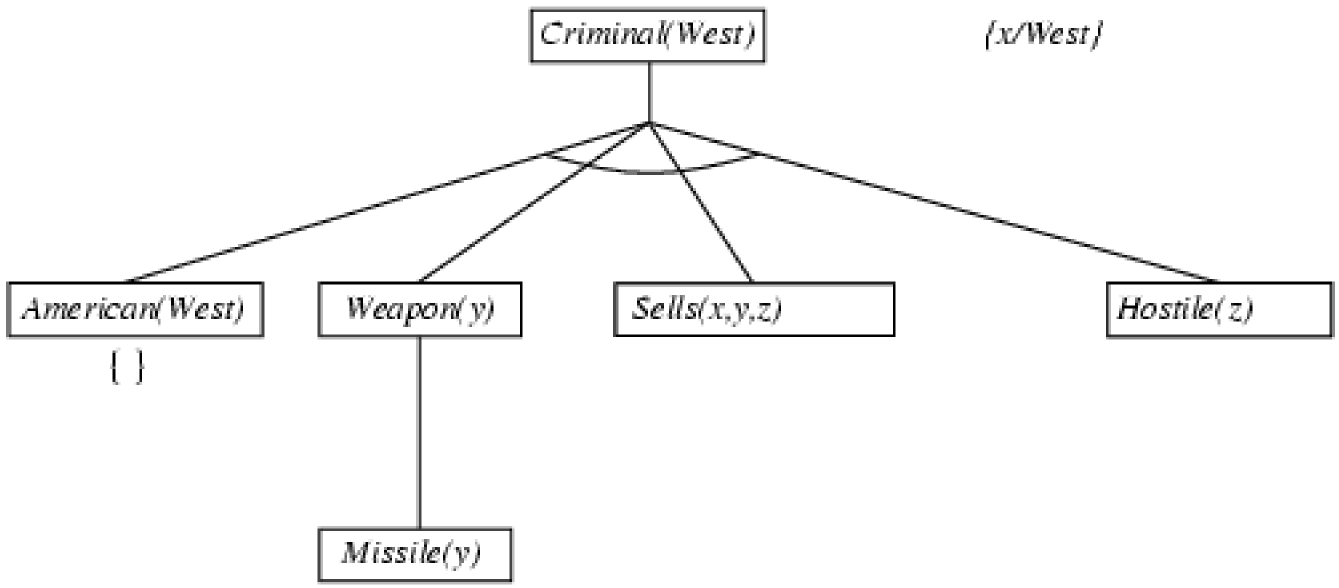
Backward chaining example



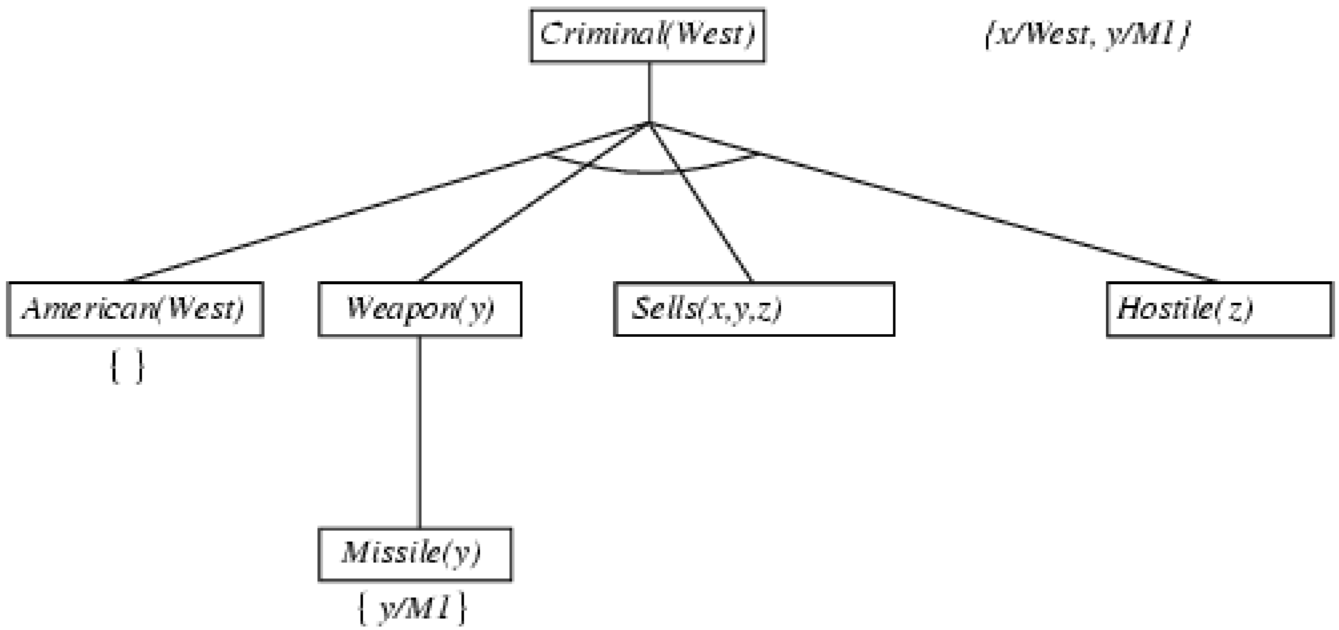
Backward chaining example



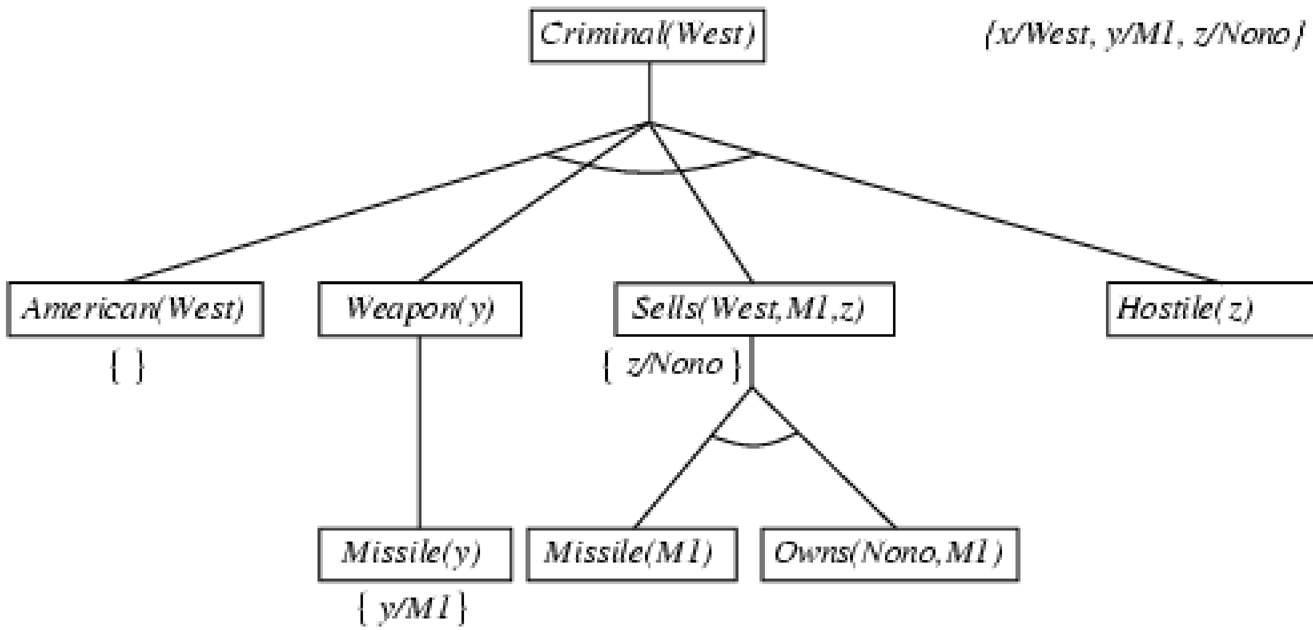
Backward chaining example



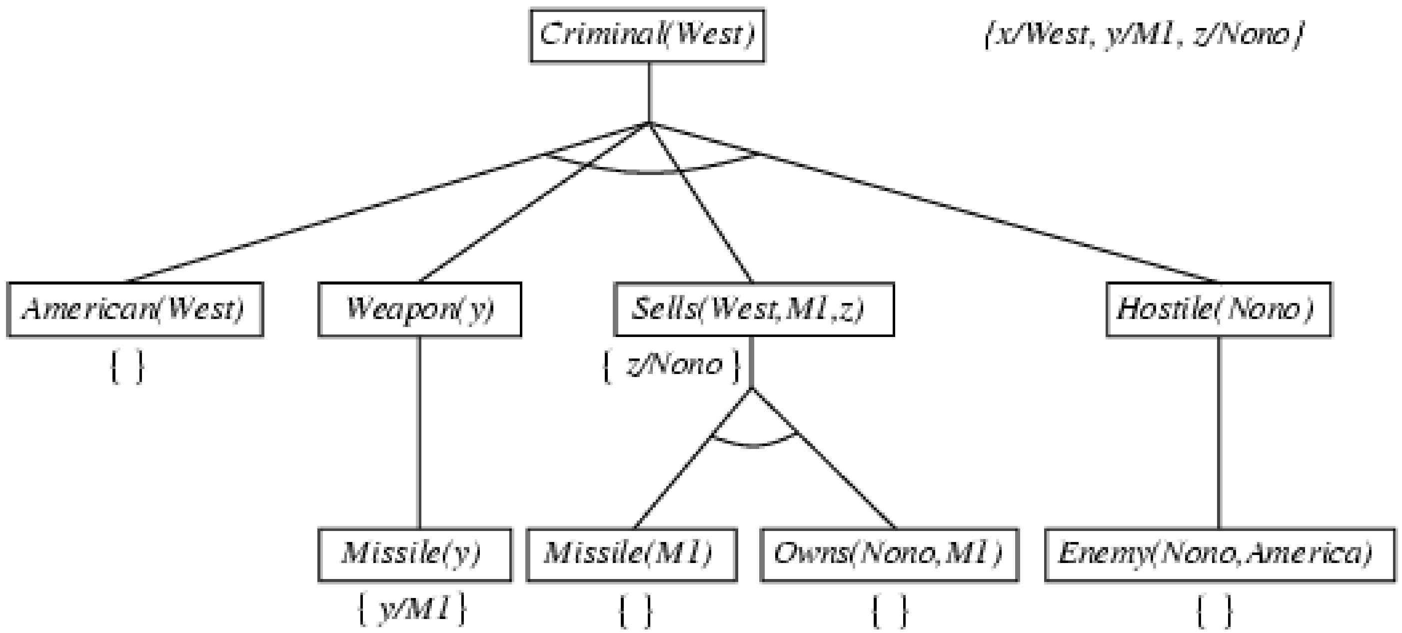
Backward chaining example



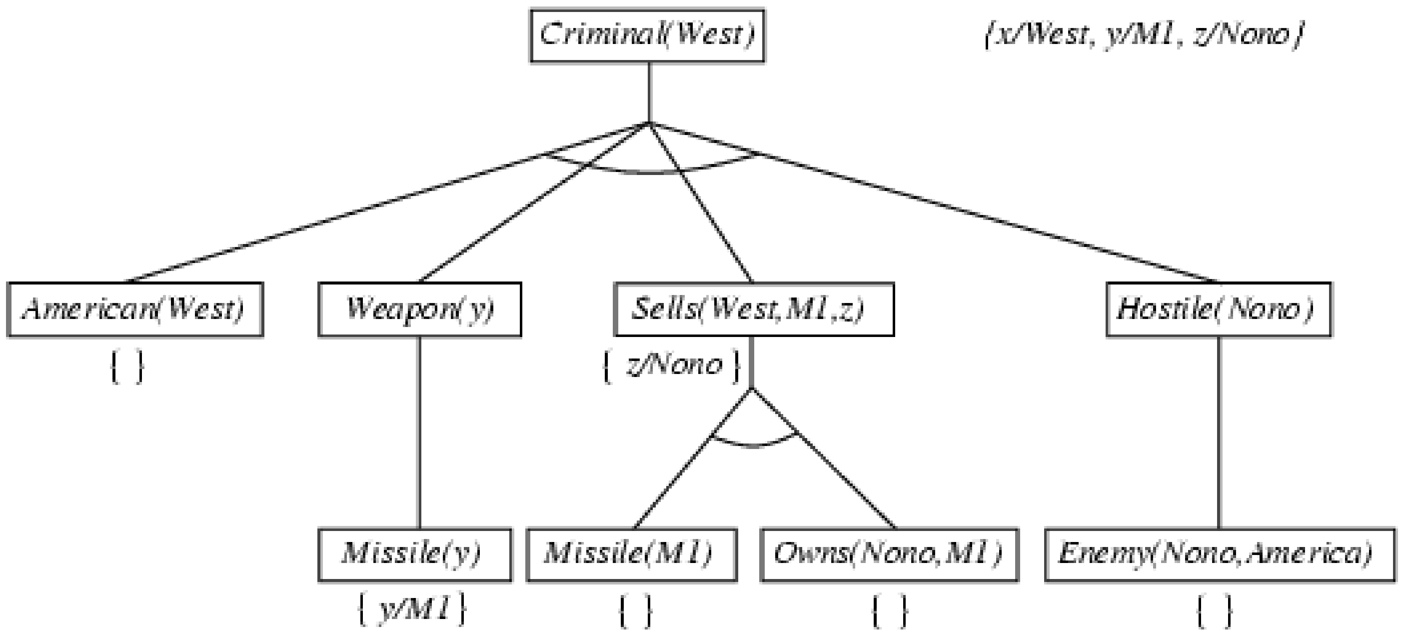
Backward chaining example



Backward chaining example



Backward chaining example



Properties of backward chaining

- Depth-first recursive proof search: space is linear in size of proof
- Incomplete due to infinite loops
  - $\Rightarrow$ fix by checking current goal against every goal on stack
- Inefficient due to repeated subgoals (both success and failure)
  - $\Rightarrow$ fix using caching of previous results (extra space)
- **Widely used for logic programming**