Generalized Modus Ponens (GMP)

$$\frac{p_1{}',\ \ p_2{}',\ \ldots,\ p_n{}',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q\sigma}$$

where $p_i{}'\sigma = p_i\sigma$ for all i

E.g. $p_1{}' =$ Faster(Bob,Pat)

$p_2{}' =$ Faster(Pat,Steve)

$p_1 \wedge p_2 \ \Rightarrow\ q \ = \ Faster(x, y) \wedge Faster(y, z) \ \Rightarrow\ Faster(x, z)$

$\sigma = \{x/Bob, y/Pat, z/Steve\}$

$q\sigma = Faster(Bob, Steve)$

GMP used with KB of <u>definite clauses</u> (*exactly* one positive literal):
either a single atomic sentence or
(conjunction of atomic sentences) $\Rightarrow$ (atomic sentence)
All variables assumed universally quantified

Soundness of GMP

Need to show that

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

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

Lemma: For any definite clause p, we have $p \models p\sigma$ by UE

1. $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \models (p_1 \wedge \ldots \wedge p_n \Rightarrow q)\sigma = (p_1\sigma \wedge \ldots \wedge p_n\sigma \Rightarrow q\sigma)$
2. $p_1{}', \ \ldots, \ p_n{}' \models p_1{}' \wedge \ldots \wedge p_n{}' \models p_1{}'\sigma \wedge \ldots \wedge p_n{}'\sigma$
3. From 1 and 2, $q\sigma$ follows by simple MP

Properties of GMP

- Why is GMP an efficient inference rule?

  - It takes bigger steps, combining several small inferences into one

  - It takes sensible steps: uses eliminations that are guaranteed to help (rather than random UEs)

  - It uses a precompilation step which converts the KB to canonical form (Horn sentences)

Horn form

Remember: sentence in Horn from is a conjunction of Horn clauses (clauses with at most one positive literal), e.g.,

$(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$, that is $(B \Rightarrow A) \wedge ((C \wedge D) \Rightarrow B)$

- We convert sentences to Horn form as they are entered into the KB using Existential Elimination and And Elimination

e.g., $\exists x\ Owns(Nono, x) \wedge Missile(x)$ becomes

Owns(Nono, M)

Missile(M)

(with M a new "skolem" symbol that was not already in the KB)

Definite Clause Form

- DEFINITE CLAUSES: are Horn clauses where there is **EXACTLY** one positive literal (Horn clauses have **AT MOST** one positive literal)
- One can often omit the universal quantifiers

- Not all sentences can be put into Definite Clause Form– in particular some variants of **negation / disjunction** cannot be put into Definite Clause Form:

$\forall x \neg$ fond_of_logic (x) ➔ no positive literal

$\forall x$ rational(x) $\vee$ crazy(x) ➔ more than one positive literal

You cannot express the above in the Definite Clause Form.

In general, not all FOL sentences that can be expressed in Horn Form or Definite Clause Form

Forward chaining

When a new fact p is added to the KB
    for each rule such that p unifies with a premise
        if the other premises are <u>known</u>
        then add the conclusion to the KB and continue chaining

Forward chaining is <u>data-driven</u>
        e.g., inferring properties and categories from percepts

Forward chaining example

Add facts 1, 2, 3, 4, 5, 7 in turn.

Number in [] = unification literal; $\surd$ indicates rule firing

1. $Buffalo(x) \land Pig(y) \;\Rightarrow\; Faster(x,y)$
2. $Pig(y) \land Slug(z) \;\Rightarrow\; Faster(y,z)$
3. $Faster(x,y) \land Faster(y,z) \;\Rightarrow\; Faster(x,z)$
4. $Buffalo(Bob)$ [1a, $\times$]
5. $Pig(Pat)$ [1b, $\surd$] $\rightarrow$ 6. $Faster(Bob, Pat)$ [3a, $\times$], [3b, $\times$]
   [2a, $\times$]
7. $Slug(Steve)$ [2b, $\surd$]
   $\rightarrow$ 8. $Faster(Pat, Steve)$ [3a, $\times$], [3b, $\surd$]
   $\rightarrow$ 9. $Faster(Bob, Steve)$ [3a, $\times$], [3b, $\times$]

Example: Forward Chaining

Current available rules

- $A \wedge C \Rightarrow E$
- $D \wedge C \Rightarrow F$
- $B \wedge E \Rightarrow F$
- $B \Rightarrow C$
- $F \Rightarrow G$

Example: Forward Chaining

Current available rules

- A ^ C => E (1)
- D ^ C => F (2)
- B ^ E => F (3)
- B => C (4)
- F => G (5)

Percept 1. A (is true)
Percept 2. B (is true)

then, from (4), C is true, then the premises of (1) will be satisfied, resulting to make E true, then the premises of (3) are going to be satisfied, thus F is true, and finally from (5) G is true.

Another Example (from Konelsky)

- Nintendo example.
  - Nintendo says it is Criminal for a programmer to provide emulators to people. My friends don't have a Nintendo 64, but they use software that runs N64 games on their PC, which is written by Reality Man, who is a programmer.

Forward Chaining

- The knowledge base initially contains:
  - 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)

Forward Chaining

$\text{Programmer}(x) \wedge \text{Emulator}(y) \wedge \text{People}(z) \wedge \text{Provide}(x,z,y) \Rightarrow$
$\text{Criminal}(x)$ (1)

$\text{Use}(\text{friends}, x) \wedge \text{Runs}(x, \text{N64 games})$

$\Rightarrow \text{Provide}(\text{Reality Man}, \text{friends}, x)$ (2)

$\text{Software}(x) \wedge \text{Runs}(x, \text{N64 games})$

$\Rightarrow \text{Emulator}(x)$ (3)

- Now we add atomic sentences to the KB sequentially, and call on the forward-chaining procedure:
  - FORWARD-CHAIN(KB, Programmer(Reality Man))

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y)
$\Rightarrow$ Criminal(x) (1)

Use(friends, x) $\wedge$ Runs(x, N64 games)
$\Rightarrow$ Provide(Reality Man, friends, x) (2)

Software(x) $\wedge$ Runs(x, N64 games)
$\Rightarrow$ Emulator(x) (3)

Programmer(Reality Man) (4)

- This new premise unifies with (1) with **subst(**{x/Reality Man}, Programmer(x)**)** but not all the premises of (1) are yet known, so nothing further happens.

Forward Chaining

$Programmer(x) \wedge Emulator(y) \wedge People(z) \wedge$
$Provide(x,z,y) \Rightarrow Criminal(x)$ (1)

$Use(friends, x) \wedge Runs(x, N64\ games)$
$\Rightarrow Provide(Reality\ Man, friends, x)$ (2)

$Software(x) \wedge Runs(x, N64\ games)$
$\Rightarrow Emulator(x)$ (3)

Programmer(Reality Man) (4)

- Continue adding atomic sentences:
  - FORWARD-CHAIN(KB, People(friends))

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$
Provide(x,z,y) $\Rightarrow$ Criminal(x) (1)

Use(friends, x) $\wedge$ Runs(x, N64 games)
$\Rightarrow$ Provide(Reality Man, friends, x) (2)

Software(x) $\wedge$ Runs(x, N64 games)
$\Rightarrow$ Emulator(x) (3)

Programmer(Reality Man) (4)

People(friends) (5)

- This also unifies with (1) with **subst(**{z/friends}, People(z)**)** but other premises are still missing.

Forward Chaining

$Programmer(x) \wedge Emulator(y) \wedge People(z) \wedge Provide(x,z,y) \Rightarrow Criminal(x)$ (1)

$Use(friends, x) \wedge Runs(x, N64\ games)$

$\Rightarrow Provide(Reality\ Man, friends, x)$ (2)

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

$\Rightarrow Emulator(x)$ (3)

Programmer(Reality Man) (4)

People(friends) (5)

- Add:
  - FORWARD-CHAIN(KB, Software(U64))

Forward Chaining

$Programmer(x) \wedge Emulator(y) \wedge People(z) \wedge Provide(x,z,y)$
$\Rightarrow Criminal(x)$ (1)

$Use(friends, x) \wedge Runs(x, N64\ games)$
$\Rightarrow Provide(Reality\ Man, friends, x)$ (2)

<u>Software(x)</u> $\wedge Runs(x, N64\ games)$
$\Rightarrow Emulator(x)$ (3)

Programmer(Reality Man) (4)

People(friends) (5)

Software(U64) (6)

- This new premise unifies with (3) but the other premise is not yet known.

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y)
$\Rightarrow$ Criminal(x) (1)

Use(friends, x) $\wedge$ Runs(x, N64 games)
$\Rightarrow$ Provide(Reality Man, friends, x) (2)

Software(x) $\wedge$ Runs(x, N64 games)
$\Rightarrow$ Emulator(x) (3)

Programmer(Reality Man) (4)

People(friends) (5)

Software(U64) (6)

- Add:
  - FORWARD-CHAIN(KB, Use(friends, U64))

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$ Criminal(x) (1)

Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Provide(Reality Man, friends, x) (2)

Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Emulator(x) (3)

Programmer(Reality Man) (4)

People(friends) (5)

Software(U64) (6)

Use(friends, U64) (7)

- This premise unifies with one of the two premises of (2) but we still don't know about the other!

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$ Criminal(x) (1)

Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Provide(Reality Man, friends, x) (2)

Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Emulator(x) (3)

Programmer(Reality Man) (4)

People(friends) (5)

Software(U64) (6)

Use(friends, U64) (7)

- Add:
  - FORWARD-CHAIN(Runs(U64, N64 games))

Forward Chaining

$Programmer(x) \wedge Emulator(y) \wedge People(z) \wedge Provide(x,z,y) \Rightarrow Criminal(x)$ (1)
$Use(friends, x) \wedge Runs(x, N64\ games) \Rightarrow Provide(Reality\ Man, friends, x)$ (2)
$Software(x) \wedge Runs(x, N64\ games) \Rightarrow Emulator(x)$ (3)

Programmer(Reality Man) (4)
People(friends) (5)
Software(U64) (6)
Use(friends, U64) (7)
Runs(U64, N64 games) (8)

- This new premise unifies with (2) and (3).

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$ Criminal(x) (1)
Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Provide(Reality Man, friends, x) (2)
Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ Emulator(x) (3)

Programmer(Reality Man) (4)
People(friends) (5)
Software(U64) (6)
Use(friends, U64) (7)
Runs(U64, N64 games) (8)

- Premises (6), (7) and (8) satisfy the implications fully.

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$ Criminal(x) (1)
Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ **Provide(Reality Man, friends, x)** (2)
Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ **Emulator(x)** (3)

Programmer(Reality Man) (4)
People(friends) (5)
Software(U64) (6)
Use(friends, U64) (7)
Runs(U64, N64 games) (8)

- So we can infer the consequents, which are now added to the knowledge base (this is done in two separate steps).

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y)$\Rightarrow$ Criminal(x) (1)

Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ **Provide(Reality Man, friends, x)** (2)

Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ **Emulator(x)** (3)

Programmer(Reality Man) (4)

People(friends) (5)

Software(U64) (6)

Use(friends, U64) (7)

Runs(U64, N64 games) (8)

Provide(Reality Man, friends, U64) (9)

Emulator(U64) (10)

- Addition of these new facts triggers further forward chaining.

Forward Chaining

Programmer(x) $\wedge$ Emulator(y) $\wedge$ People(z) $\wedge$ Provide(x,z,y) $\Rightarrow$ Criminal(x)	(1)
Use(friends, x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ **Provide(Reality Man, friends, x)**	(2)
Software(x) $\wedge$ Runs(x, N64 games) $\Rightarrow$ **Emulator(x)**	(3)
Programmer(Reality Man)	(4)
People(friends)	(5)
Software(U64)	(6)
Use(friends, U64)	(7)
Runs(U64, N64 games)	(8)
Provide(Reality Man, friends, U64)	(9)
Emulator(U64)	(10)
Criminal(Reality Man)	(11)

- Which results in the final conclusion: Criminal(Reality Man)

Forward Chaining

- Forward Chaining acts like a breadth-first search at the top level, with depth-first sub-searches.
- Since the search space spans the entire KB, a large KB must be organized in an intelligent manner in order to enable efficient searches in reasonable time.

Forward chaining algorithm

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*

  repeat until *new* is empty
    new $\leftarrow \{\}$
    for each sentence r **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
      for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or *new* **then do**
          add q' to *new*
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *fail* **then return** ϕ
    add *new* to KB
  return *false*

Standardize-Apart : replaces all variables in the arguments with NEW variables that will not conflict with any other variables used

Example Knowledge Base

... it is a crime for an American to sell weapons to hostile nations:

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

Nono … has some missiles, i.e., $\exists x\ Owns(Nono,x) \wedge Missile(x)$:

$Owns(Nono,M_1)$ and $Missile(M_1)$

… all of its missiles were sold to it by Colonel West

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

Missiles are weapons:

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

An enemy of America counts as "hostile“:

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

West, who is American …

$American(West)$

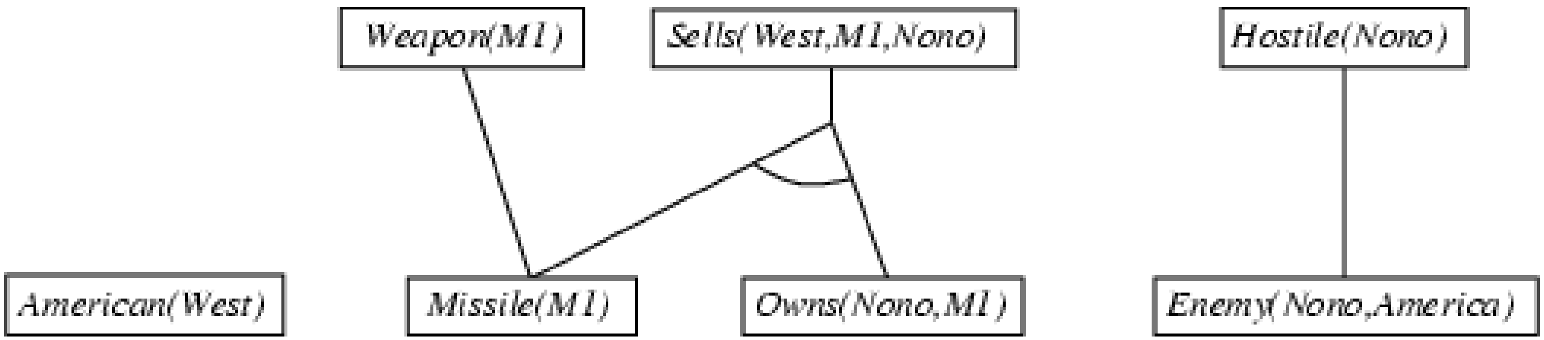
The country Nono, an enemy of America …

$Enemy(Nono,America)$

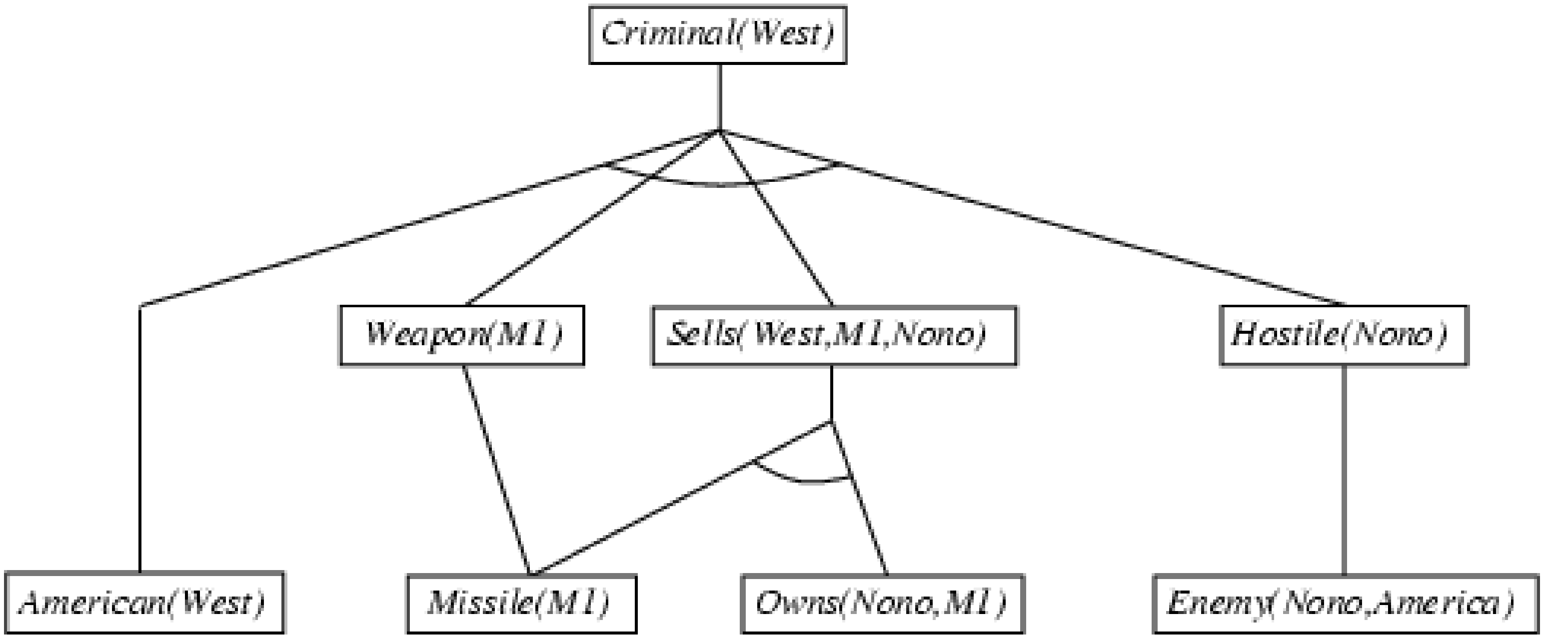
Forward chaining proof

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

Forward chaining proof



Forward chaining proof



Properties of forward chaining

- Sound and complete for first-order **definite** clauses

- Datalog = first-order definite clauses + **no functions**
- FC terminates for Datalog in finite number of iterations

- May not terminate in general **if α is not entailed**

- This is unavoidable: entailment with definite clauses is semidecidable – if a sentence is entailed, FC will eventually terminate; but non-termination is not evidence of non-entailment

Efficiency of forward chaining

Incremental forward chaining: no need to match a rule on iteration *k* if a premise wasn't added on iteration *k-1*

$\Rightarrow$ match each rule whose premise contains a newly added positive literal

Matching itself can be expensive:

Database indexing allows O(1) retrieval of known facts

- e.g., query *Missile(x)* retrieves *Missile*(M_1)

Forward chaining is widely used in **deductive databases**