Useful Equivalencies

[Needs only $\&$ $\neg$ $\forall$]

$$\neg\neg P \equiv P$$

$$P \vee Q \equiv \neg[(\neg P) \wedge (\neg Q)]$$

$$\neg[P \vee Q] \equiv (\neg P) \wedge (\neg Q)$$

$$P \Rightarrow Q \equiv (\neg P) \vee Q$$
$$\equiv \neg(P \wedge \neg Q)$$

$$P \Leftrightarrow Q \equiv [(P \Rightarrow Q) \wedge (Q \Rightarrow P)]$$
$$\equiv \neg(P \wedge \neg Q) \wedge \neg(Q \wedge \neg P)$$

$$\exists x.\, \phi(x) \equiv \neg[\forall x. \neg\phi(x)]$$

$$\neg[\exists x.\, \phi(x)] \equiv \forall x. \neg\phi(x)$$

$$\varphi \Rightarrow \tau \equiv \neg\tau \Rightarrow \neg\varphi$$

$$\exists! x.\, \phi(x) \equiv \exists x.\, [\phi(x) \wedge \forall z.\, \phi(z) \Rightarrow z = x]$$

...Exactly n values of φ ...

Example of $\Rightarrow$, $\forall$

- Using $\mathcal{U}$ = Set of natural numbers, $\mathcal{N}$

- $\forall n.\, 6|n \;\Rightarrow\; 2|n$
  $\equiv \quad \forall n.\, \neg 6|n \;\vee\; 2|n$

- $$A = \{\; n : \neg 6|n \;\} = \{1,2,3,4,5,\;\; 7,8,9,10,11,\;\; 13,14,\ldots\}$$
  $$B = \{\; n : 2|n \;\} = \{\; 2,\; 4,\; 6,\; 8,\; 10,\; 12,\; 14,\ldots\}$$

- Notice $A \cup B = \mathcal{N}$
  (Hence each $n \in \mathcal{N}$, n satisfies either $\neg 6|n$ or $2|n$)

  So $\quad \forall n\; 6|n \;\Rightarrow\; 2|n$

Gödel's Incompleteness Proof

- $\exists$ true Sentences that cannot be proved

- Consider Numbers:

  0, succ, +, $\times$, . . .
  $+$ axioms

  Can enumerate all syntactically legal sentences
  $\Rightarrow$ give each sentence α, a number $G(\alpha) \in \mathcal{N}$
  $\Rightarrow$ give each PROOF $\langle \alpha_i \rangle$, a number $G(\langle \alpha_i \rangle)$

- Let $A =$ set of true statements about $\mathcal{N}$

  Let $\alpha(j, A) \quad \Leftrightarrow$
  $\forall i$ i is NOT Gödel number of proof, using A, of $G^{-1}(j)$

  Let $\quad \sigma = \alpha(\, G(\sigma),\, A)$
  ie, "I am not provable, from A"

- If σ is provable from A, then σ is FALSE
  $\Rightarrow$ A inconsistent!

  As A consistent,
  $\Rightarrow$ σ NOT provable from A
  $\Rightarrow$ σ is true statement!

$\Rightarrow$ σ is true, but cannot be proven from A!

1c. How to Compute (> 174 50)?

Challenge: Determine truth of (> 174 50)

Option 1: Explicitly store

```
(>   51 50)   (>   52 50)   (>   53 50)   (>   54 50)
(>   55 50)   (>   56 50)   (>   57 50)   (>   58 50)
                        ...
(>  173 50)   (>  174 50)   (>  175 50)   (>  176 50)
                        ...
(> 2021 50)   (> 2022 50)   (> 2023 50)   (> 2024 50)
                        ...
```

and negative facts:

```
¬(> 41 50)   ¬(> 42 50)   ¬(> 43 50)   ¬(> 44 50)
...
```

as well as

```
(> 1 0)   (> 2 0)   (> 3 0)   (> 4 0)   ...
          (> 2 1)   (> 3 1)   (> 4 1)   ...
                    (> 3 2)   (> 4 2)   ...
                              (> 4 3)   ...
                                        ...
```

- Requires $\approx \infty$ storage!

Is there a better way?

Option 2: Procedural Attachment

- To compute (> x y),

  Use procedure FetchGT

  where FetchGT returns nil or t

- FetchGT(σ: proposition)

```
    (if  (> (cadr σ) (caddr σ))
         t
         nil )
```

Eg:
```
-> (FetchGT '(> 174 50))
t
-> (FetchGT '(> 23 41))
()
```

Procedural Attachment: +

- Find `w` s.t. `(+ 10 65 w)`

- Explicit storage: ∞ space!

- Procedure:
  To compute `(+ 10 65 w)`
  Use procedure **FetchPlus**
  where **FetchPlus** returns appropriate binding list:

```
FetchPlus( σ:  proposition )
  (Match (cadddr σ) (+ (cadr σ) (caddr σ)) )
;;;        w            10         65
```

```
(FetchPlus (+ 10 65 w))   ↦ (w/75)
(FetchPlus (+ 10 65 75))  ↦ t
(FetchPlus (+ 10 65 921)) ↦ ()
```

- MRS Solution:
  a) MetaTell `(ToFetch (> &x &y) FetchGT)`
     MetaTell `(ToFetch (+ &x &y &z) FetchPlus)`
  b) MetaTell `(relnproc > >)`
     MetaTell `(funproc + +)`

Procedural Attachment

Why? (Space) inefficient to store explicitly.

What? Use procedure to solve query.

Constraints: *Sound* procedure
?Only some bound-sets (directions)?

Eg: $<$, `+`, `Sort`, . . .

Gen'l: MRS allows user to define
how to answer arbitrary `Ask`ed proposition

Declarative/Procedural

- Axioms *[eg "`man(X) :- human(X), male(X).`"]*
  have two readings:

  declarative: For any `X`,
  if `human(X)` and `male(X)` are true,
  then so is `man(X)`.

  procedural: For any `X`,
  to achieve goal `man(X)`,
  it is sufficient to achieve
  `human(X)` and `male(X)`.

- Like procedure: `X` ≈ formal parameter

`man(X)`	head	
`human(X), male(X)`	body	can fail/succeed

- Goal of `man(a)` ≈ call with `X` ← `a`

[see top-down theorem proving, [Reiter] p19-20]

Top Down Theorem Proving qua Procedure Calling

Consider goal $\texttt{G} = \boxed{(\texttt{g}_1,\ \ldots,\ \texttt{g}_n)}$
as procedural calls (performed left-to-right):

Goal $\boxed{(\texttt{g}_1,\ \ldots,\ \texttt{g}_n)}$ succeeds (with $\sigma_1 \circ \sigma$) if

1.
   - $\boxed{\texttt{g}_1}$ succeeds (with σ_1), and
   - $\boxed{(\texttt{g}_2\sigma_1,\ \ldots,\ \texttt{g}_n\sigma_1)}$ succeeds (with σ)

2. Literal $\boxed{\texttt{g}}$ succeeds (with σ) if either
   - g unifies with atomic "procedure" t (under σ)
   - g unifies with the head of "procedure"
     $\texttt{t :- t}_1\texttt{, ..., t}_m$
     under σ', and
     $\boxed{(\texttt{t}_1\sigma',\ \ldots,\ \texttt{t}_m\sigma')}$ succeeds (with σ_m).
     $[\sigma = \sigma' \circ \sigma_m]$

[If sucessful, Prolog returns unifer σ.]

Different from Standard Procedures

1. Success *vs* Fail

$$KB = \left\{ \begin{array}{l} \text{p(X) :- } \langle \texttt{body1} \rangle. \\ \text{p(X) :- } \langle \texttt{body2} \rangle. \end{array} \right\}$$

"Called" on **`p(a)`**:

≈ Procedure: try ⟨**body1**⟩ and if that fails,
try ⟨**body2**⟩ .

2. Automatic backtracking (all sol'ns, 1-by-1)

$$KB = \left\{ \begin{array}{lll} \text{q(a,b)} & \text{q(a,c).} & \text{q(a,d).} \\ \text{r(c).} & \text{r(d).} & \\ \multicolumn{3}{l}{\text{p(X) :- q(X,Y), r(Y).}} \end{array} \right\}$$

"Called" on **`p(a)`**:

assign `Y=b` then fails

re-assign `Y=c` then succeeds

{ *If Prolog asked for next proof:* }

re-assign `Y=d` then succeed

{ *If Prolog asked for next proof ⇒ fail & done.* }

Differences (con't)

3. Non-Determinism (in principle)
   Can "execute" a goal in > 1 way

$$KB = \left\{ \begin{array}{l} \text{p(X,a) :- } \langle \texttt{body1} \rangle. \\ \text{p(X,Y) :- } \langle \texttt{body2} \rangle. \end{array} \right\}$$

   Called on `(..., p(b,Z), ...)`

   Will it execute $\langle$**body1**$\rangle$?
   Will it get to $\langle$**body2**$\rangle$?

   (Made deterministic by Depth-First strategy)

4. Variables
   Need *not*
   - be given values for procedural call
   - get values from procedure

   ...but constrained wrt other variables
   via unification

   (Major strength of Prolog.)

Examples of Variable Use

1. Constrained by goal:

   $$KB = \left\{ \begin{array}{llll} \text{p(X,Y) :- q(X), r(Y).} & & & \\ \text{q(b).} & \text{r(a).} & \text{r(b).} & \text{r(c).} \end{array} \right\}$$

   Goal `p(Z,Z)`

   does NOT give values to args (`X`, `Y`)

   but does constrain them to be equal.

   So succeeds only with $\{ {}_{\mathtt{b}}Z \}$

2. $KB = \{$ "`r(s(X),Y) :- ....`", $\cdots \}$

   Goal: `r(Z,p(Z))`

   forces

   `Z` to be `s(···)`, and

   `Y` to be `p(Z)` = `p(s(···))`

   where "···" determined by later goals.

   (Using MGU ≈ least commitment:

   establishes only what HAS to be true.)