We present another example illustrating Herbrand models. Consider the following program P:

$$likes(john, X) \leftarrow likes(X, apples)$$

$$likes(mary, apples) \leftarrow$$

Suppose the language $\mathcal{L}$ contained no symbols other than those in P. Then, $\mathcal{B}(P)$ is the set $\{likes(john, john), likes(john, apples), likes(apples, john), likes(john, mary), likes(mary, john), likes(mary, apples), likes(apples, mary), likes(mary, mary), likes(apples, apples)\}$. Now, $\{likes(mary, apples), likes(john, mary)\}$ is a subset of $\mathcal{B}(P)$, and is a Herbrand interpretation. Moreover, it is also a Herbrand model for P. Similarly, $\{likes(mary, apples), likes(john, mary), likes(mary, john)\}$ is also a model for P. The ground instantiation $\mathcal{G}(P)$ for this program is:

$$likes(john, john) \leftarrow likes(john, apples)$$

$$likes(john, mary) \leftarrow likes(mary, apples)$$

$$likes(john, apples) \leftarrow likes(apples, apples)$$

$$likes(mary, apples) \leftarrow$$

G(P)

It can be verified[25] that the interpretation $\{likes(mary, apples), likes(john, mary)\}$ is a model for the $\mathcal{G}(P)$ above.

Q: Is every model for G(P) a model for P?

Theorem 16 *A clausal formula Σ has a model if and only if its ground instantiation $\mathcal{G}(\Sigma)$ has a Herbrand model.*

Proof: $\Rightarrow$: Suppose Σ has a model M. Then we define the following Herbrand interpretation I as follows. Let P be an n-ary predicate symbol occurring in Σ. Then we define the function I_P from U_L^n to $\{T, F\}$ as follows: $I_P(t_l, \ldots, t_n) = T$ if $P(t_1, ..., t_n)$ is true under M, and $I_P(t_1, ..., t_n) = F$ otherwise. It can easily be shown that $I = \cup_{P \in \Sigma} I_P$ is a Herbrand model of Σ.

$\Leftarrow$: This is obvious (a Herbrand model is a model). $\square$

$t_1, t_2 \ldots t_n \in U_L$

Rem: M is a set of atoms. (ground).

$$\begin{array}{ccc} \mathcal{G}(\Sigma) & \cdots\cdots & MM(\mathcal{G}(\Sigma)) \\ & & \Downarrow \\ \Sigma & \cdots\cdots & M_\Sigma \end{array}$$

smallest, and is the minimal model. There is an important result relating a definite clausal formula Σ, its minimal model $\mathcal{MM}(\Sigma)$ and the ground atoms that are logical consequences of Σ:

Theorem 17 *If α is a ground atom then $\Sigma \models \alpha$ if and only if $\alpha \in MM(\Sigma)$.*

same as $MM(G(\Sigma))$.

Here $MM(\cdot)$ denotes the minimal model. Thus, the minimal model of a definite clausal formula is identical to the set of all ground atoms logically implied by that formula. Thus, the minimal model provides, in effect, denotes the meaning (or semantics) of the formula. The proof of this theorem follows nearly from theorem 12 that was proved earlier.

From prop logic:

$G(\Sigma) \models \alpha$ iff $\alpha \in MM(G(\Sigma)) \equiv MM(\Sigma)$

To find min'l-model of Σ, .. use T_Σ to compute $MM(G(\Sigma))$

Found using DPLL

MLNs use any model of $G(\Sigma)$

BLPs use $MM(G(\Sigma))$

The statement *"Any animal that has hair is a mammal"* can be written as a clause using monadic predicates (*i.e.* predicates with arity 1):

$$\forall X \ is\_mammal(X) \leftarrow has\_hair(X)$$

Usually clauses are written without explicit mention of the quantifiers:

$$is\_mammal(X) \leftarrow has\_hair(X)$$

$$is\_mammal(X) \leftarrow has\_milk(X)$$

$$is\_bird(X) \leftarrow has\_feathers(X)$$

$$\ldots$$

Got almost close to prop logic
· with skolemization
&
· implicit $\forall$.

Datalog

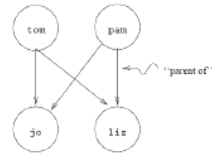
Datalog is a subset of the language of first order language; it has all the components of first order logic (variables, constants and recursion), except functions. A Datalog "expert" system will encode these rules using monadic predicates as:

```
is_mammal(X) :- has_hair(X).
is_mammal(X) :- has_milk(X).
is_bird(X) :- has_feathers(X).
is_bird(X) :- can_fly(X), has_eggs(X).
is_carnivore(X) :- is_mammal(X), eats_meat(X).
is_carnivore(X) :- has_pointed_teeth(X), has_claws(X), has_pointy_eyes(X).
cheetah(X) :- is_carnivore(X), has_tawny_colour(X), has_dark_spots(X).
tiger(X) :- is_carnivore, has_tawny_colour(X), has_black_stripes(X).
penguin(X) :- is_bird(X), cannot_fly(X), can_swim(X).
```

Now here are some statements[26] particular to animals:

```
has_hair(peter).
has_green_eyes(peter).
eats_meat(peter).
has_milk(bob).
has_tawny_colour(bob).
can_fly(bob).
fat(peter).
has_tawny_colour(peter).
has_black_stripes(peter).
eats_meat(bob)
has_dark_spots(bob).
```

$$
\begin{aligned}
&parent(tom, jo) \leftarrow \\
&parent(pam, jo) \leftarrow \\
&parent(tom, liz) \leftarrow \\
&parent(pam, liz) \leftarrow
\end{aligned}
$$



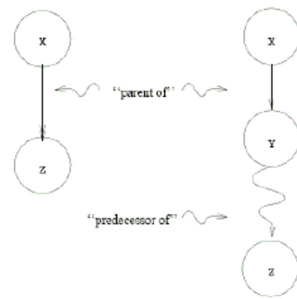
Consider the *predecessor* relation, namely, all ordered tuples $< X, Y >$ *s.t.* X is an ancestor of Y. This set will include Y's parents, Y's grandparents, Y's grandparents' parents, etc.

$$
\begin{aligned}
pred(X, Y) &\leftarrow parent(X, Y) \\
pred(X, Z) &\leftarrow parent(X, Y), parent(Y, Z) \\
pred(X, Z) &\leftarrow parent(X, Y1), parent(Y1, Y2), parent(Y2, Z) \\
&\ldots
\end{aligned}
$$

Implicit $\exists Y$

As can be seen through this example, variables and constants are not enough: we need *recursion*:

$\forall X, Z$ X is a predecessor of Z if
1. X is a parent of Z; or
2. X is a parent of some Y, and Y is a predecessor of Z



The predecessor relation is thus

$$
\begin{aligned}
pred(X, Y) &\leftarrow parent(X, Y) \\
pred(X, Z) &\leftarrow parent(X, Y), pred(Y, Z)
\end{aligned}
$$

$\equiv$

$\forall x \forall y \forall z \, (pred(x, z) \leftarrow parent(x, y) \, pred(y, z))$

Prolog = Predicates + Variables + Constants + Functions

Consider Peano's postulates for the set of natural numbers $\mathcal{N}$.

1. The constant 0 is in $\mathcal{N}$
2. if X is in $\mathcal{N}$ then $s(X)$ is in $\mathcal{N}$
3. There are no other elements in $\mathcal{N}$
4. There is no X in $\mathcal{N}$ s.t. $s(X) = 0$
5. There are no X, Y in $\mathcal{N}$ s.t. $s(X) = s(Y)$ and $X \neq Y$

We can write a definite clause definition using 1 constant symbol and 1 unary function symbol for enumerating the elements of $\mathcal{N}$:

$$
\begin{array}{ll}
natural(0) \leftarrow & 1 \\
natural(s(X)) \leftarrow natural(X) & 2
\end{array}
$$

The elements of $\mathcal{N}$ can be now generated by asking:

natural(N)?

$s(X) = X+1$

natural(Y) ← natural(X), Y is X+1.

← natural(10)

↓ 2

← natural(9)

↓ 2

⋮

← natural(0)

↓ 1

□

Prolog also supports lists. Lists are simply collections of objects. For e.g. $1, 2, 3 \ldots$ or $1, a, dog, \ldots$. Lists are defined as follows:

1. The constant nil is a list
2. If X is a term, and Y is a list then $.(X, Y)$ is a list

So the list $1, 2, 3$ is represented as:

$$.(1, .(2, .(3, nil)))$$

Usually logic programming systems use a "[" "]" notation, in which the constant nil is represented as [] and the list $1, 2, 3$ is $[1, 2, 3]$. In this notation, the symbol | is used to separate a list into a "head" (the elements to the left of the |) and a "tail" (the list to the right of the |). Thus:

List	Represented as	Values of variables
$[1, 2, 3]$	$[X\|Y]$	$X = 1, Y = [2, 3]$
$[[1, 2], 3]$	$[X\|Y]$	$X = [1, 2], Y = [3]$
$[1]$	$[X\|Y]$	$X = 1, Y = []$
$[1\|2]$	$[X\|Y]$	$X = 1, Y = 2$
$[1]$	$[X, Y]$	
$[1, 2, 3]$	$[X, Y\|Z]$	$X = 1, Y = 2, Z = [3]$

Consider the following set of clauses S:

$$\begin{array}{l} likes(john, flowers) \leftarrow \\ likes(mary, food) \leftarrow \\ likes(mary, wine) \leftarrow \\ likes(john, wine) \leftarrow \\ likes(john, mary) \leftarrow \\ likes(paul, mary) \leftarrow \end{array}$$

likes(X, food)

If you entered these clauses into a program capable of executing logic programs (some implementation of Prolog), and asked:

$$likes(john, X)?$$

you will get a number of answers:

$$\begin{array}{l} X = flowers \\ X = wine \\ X = mary \end{array}$$

On the other hand, if the query were

$$likes(john, X), likes(mary, X)?$$

the answer should be:

$$X = wine$$

Broad sense:- Resolution
↳ Look for complementary literals

Conditional	Clausal Form
$\forall x(Ape(x) \leftarrow Human(x))$	$\forall x(Ape(x) \vee \neg Human(x))$
$Human(fred) \leftarrow Human(father(fred))$	$Human(fred) \vee \neg Human(father(fred))$

For resolution to apply, we require the clausal forms to contain a pair of complementary literals. We nearly do have such a pair: $\neg Human(x)$ in the first clause and $Human(fred)$ in the second. It is apparent that if variable x in the first clause were to be restricted to the term $fred$, then we would indeed have a complementary pair, and the resolvent is:

$x \rightarrow fred$. Unifier -

OR

$x/fred$

Resolvent	Clausal Form
$Ape(fred) \leftarrow Human(father(fred))$	$Ape(fred) \vee \neg Human(father(fred))$

A single resolution step in predicate logic thus involves 'substituting' terms for variables so that a complementary pair of literals results. Here, such a pair would result if we could somehow 'match' the literals $Human(x)$ and $Human(fred)$. The resulting mapping of variables to terms is called the *unifier* of the two literals. Thus, mapping x to $fred$ is a unifier for the literals $Human(x)$ and $Human(fred)$.

Unifier is a particular kind of substitution

Substitution

1. They should be *functions*. That is, each variable to the left of the / should be distinct. Thus, $\{x/fred, x/bill\}$ is not a legal substitution; and

2. They should be *idempotent*. That is, each term to the right of the / should not contain a variable that appears to the left of the /. Thus, $\{x/father(x)\}$ is not a legal substitution. This test is sometimes called the "occurs-check". The occur-check disallows self-referential bindings such as $X/f(X)$. However, the temptation to omit the occur-check in unification algorithms is very strong, owing to the high processing cost of including it; it is the only test in the comparison cycle which has to scrutinize the inner contents of terms, whereas all other tests examine only the terms' principal (outermost) symbols.

$\theta: \{v_1/t_1, v_2/t_2, \ldots v_n/t_n\}$

must be variables

must be terms

$\theta := \{x/fred\}$ $\quad \alpha = \forall x(Ape(x) \vee \neg Human(x))$

Then $\quad \alpha\theta = (Ape(fred) \vee \neg Human(fred))$

Substitutions can be composed

A pair of substitutions can be *composed* ('joined together'). For example, composing $\{x/father(y)\}$ with $\{y/fred\}$ results in $\{x/father(fred)\}$. In general, the result of composing substitutions

$$\theta_1 = \{u_1/s_1, \ldots, u_m/s_m\}$$

$$\theta_2 = \{v_1/t_1, \ldots, v_n/t_n\}$$

is (this may not be a legal substituition): Why?

$$\theta_1 \circ \theta_2 = \{u_1/s_1\theta_2, \ldots, u_m/s_m\theta_2\} \cup \{v_i/t_i | v_i \notin \{u_1, \ldots, u_m\}\}$$

Order of application in $(\alpha\, \theta_1 \circ \theta_2)$

First θ_1 & then θ_2 on $\alpha\theta_1$

Proof sketch: The proof for this example is easy: suppose I is a model, with domain D, of $P(x) \vee \neg Q(y)$. Then for all $d_1 \in D$, and for all $d_2 \in D$, $I_P(d_1) = T$ or $I_Q(d_2) = F$. Suppose a is mapped to domain element d by I, then for all $d \in D$, $I_P(d) = T$ or $I_Q(d) = F$. Hence I is a model of $P(a) \vee \neg Q(y)$. It is clear that for different α or θ, a similar proof can always be given. Hence always $\alpha \models \alpha\theta$. $\Box$

Remember that $\{x/a\}$ should subst all x with a.

$d' \in D$

either $P \to d_1$ or $Q \to d_2$ for all $d_1, d_2 \in D$

$\Rightarrow$ either $P \to d'$ or $Q \to d_2$ for all $d_2 \in D$

We are now in a position to state more formally the notion of unifiers. To say that a substitution θ is a unifier for formulæ α_1 and α_2 means $\alpha_1\theta = \alpha_2\theta$. However, there can be many unifiers. For example, the formulæ $\alpha_1 : \forall x \forall z Parent(father(x), z)$ and $\alpha_2 : \forall y Parent(y, fred)$ have as unifiers $\theta_1 = \{x/fred, y/father(fred), z/fred\}$ and $\theta_2 = \{y/father(x), z/fred\}$. In the first case $\alpha_1\theta_1 = \alpha_2\theta_1 = Parent(father(fred), fred)$; and in the second case $\alpha_1\theta_2 = \alpha_2\theta_2 = \forall x Parent(father(x), x)$. Notice that θ_2 is, in some sense, more 'general' than θ_1 as it imposes less severe constraints on the variables. There is, in fact, a *most general unifier* (or mgu) for a pair of formulæ. The substitution θ is a most general unifier for α_1 and α_2 if and only if:

& mgu

1. $\alpha_1\theta = \alpha_2\theta$ (that is, θ is a unifier for α_1 and α_2); and

2. For any other unifier σ for α_1 and α_2, there is a substitution μ such that $\sigma = \theta \circ \mu$ (that is, $\alpha_1\sigma$ is a substitution instance of $\alpha_1\theta$).

Verify

$\nexists\ \theta_3$ s.t.

$\theta_2 = \theta_3 \cdot \mu$

mgu.

θ_2 is an mgu.

$\theta_1 = \theta_2 \circ \mu$ $\qquad \mu = ?$ $\qquad \mu = \{x/fred\}$

$\{x/fred, y/father(fred), z/fred\}$ $\qquad \{y/father(x), z/fred\}$

Extending Resolution from 0 order to 1st order

1. Rename all variables in clause C_2 so that they cannot be confused with those in C_1 (for the variables in C_2 are independent of those in C_1 and the renamed clause is equivalent to C_2). This is sometimes called "standardising the clauses apart";

2. Identify complementary literals and see if an mgu exists;

3. Apply mgu and form the resolvent C.

Previous slide

0 order logic

act on diff vars.

So that if $C_1\theta_1 = C_2\theta_2$

then $C_2\theta_1 = C_2$

$C_1\theta_2 = C_1$

Formula	Clausal Form
$C_1 : \forall x(Ape(x) \leftarrow Human(x))$	$\forall x(Ape(x) \vee \neg Human(x))$
$C_2 : \forall x(Human(x) \leftarrow Human(father(x)))$	$\forall x(Human(x) \vee \neg Human(father(x)))$

Example:

1. $C_1 :- \forall x\,(Ape(x) \vee \neg Human(x))$

   $C_2 :- \forall y\,(Human(y) \vee \neg Human(father(y)))$

2. $\neg Human(x)$ & $Human(y)$

   mgu = $\theta_1 = \{x/y\}$ $\theta_2 = \epsilon$

3. $C_1\theta_1 = \forall y\,(Ape(y) \vee \neg Human(y))$

   $\forall y\,(Human(y) \vee \neg Human(father(y)))$

   $C_2\theta_2 = C_2$

   Resolvent : ... (y))

As with propositional logic, the set-based notation used for clauses (page 60) allows us to present resolution in a compact (algebraic) form:

$$R = (C_1 - \{L\})\theta \cup (C_2 - \{M\})\theta$$

- FACTORING

$C_1: \forall x \forall y\ (Human(x) \vee Human(y))$

$C_2: \forall u \forall v\ (\neg Human(u) \vee \neg Human(v))$

Write in words & verify that $(C_1 \cup C_2)$ is unsatisfiable.

Unfortunately :- Resolution as described does not lead to $\Box$

Every step you will get Resolvent :- $\forall p \forall q\ (Human(p) \vee \neg Human(q))$.

$\therefore$ Need to eliminate redundant literals = Factoring.

Factor$(C_1) = \forall x\ (Human(x))$

Factor$(C_2) = \forall y\ (\neg Human(y))$

Key to factoring :- Find mgu for literals in C

Formally, if C is a clause, $L_1, \ldots, L_n$ $(n \geq 1)$ some unifiable literals from C, and θ an mgu for the set $\{L_1, \ldots, L_n\}$, then the clause obtained by deleting $L_2\theta, \ldots, L_n\theta$ from $C\theta$ is called a *factor* of C. For example, $Q(a) \vee P(f(a))$ is a factor of the clause $\neg Q(a) \vee P(f(a)) \vee P(y)$ using $\{y/f(a)\}$ as an mgu for $\{P(f(a)), P(y)\}$. Also, $Q(x) \vee P(x,a)$ is a factor of $Q(x) \vee Q(y) \vee Q(x) \vee P(x,a)$.

Combinatorial? No..

$L_1 = \neg Q(a)$ $\quad L_2 = P(f(a))$ $\quad L_3 = P(y)$

$mgu = \theta = \{y / f(a)\}$

$C \xrightarrow{\text{factoring}} C'$

$\underline{\underline{Q}}$: Is $C \equiv C'$? $\Big\}$ $\therefore C \models C\theta \equiv C'$

$C' = C\theta$

Like skolemization

But sufficient for resdn.