Announcements

- QA session next week on Zoom (see Piazza)

What is now required is to give the greatest possible development to mathematical logic, to allow to the full the importance of relations... If this can be successfully accomplished, there is every reason to hope that the near future will be as great an epoch in pure philosophy as the immediate past has been in the principles of mathematics. Great triumphs inspire great hopes; and pure thought may achieve, within our generation, such results as will place our time, in this respect, on a level with the greatest age of Greece.

– Bertrand Russell, Mysticism and Logic and Other Essays [1917]

Since Last midterm

- difference lists, definite clause grammars and natural language interfaces to databases
- computer algebra and calculus
- Triples are universal representations of relations, and are the basis for RDF, and knowledge graphs
- URIs/IRIs provide constants that have standard meanings
- Ontologies define the meaning of symbols used in information systems.
- You should know what the following mean: RDF, IRI, rdf:type, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range
- Complete knowledge assumption and negation as failure
- Extra-logical predicates
- Substitutions and Unification

Today

- Proofs and answers. Negation with variables.

Unifiers

- Substitution σ is a unifier of e_1 and e_2 if $e_1\sigma = e_2\sigma$.
- Substitution σ is a most general unifier (mgu) of e_1 and e_2 if
  - σ is a unifier of e_1 and e_2 and
  - if substitution σ' also unifies e_1 and e_2, then $e\sigma'$ is an instance of $e\sigma$ for all atoms e.
- If two atoms have a unifier, they have a most general unifier.
- If there are more than one most general unifiers, they only differ in the names of the variables.

Top-down Propositional Proof Procedure (recall)

- Idea: search backward from a query to determine if it is a logical consequence of KB.
- An answer clause is of the form:

$$yes \;\text{:-}\; a_1,\; a_2,\; \ldots,\; a_m$$

- The (SLD) resolution of this answer clause on atom a_1 with the clause in the knowledge base:

$$a_1 \;\text{:-}\; b_1,\; \ldots,\; b_p$$

  is the answer clause

$$yes \;\text{:-}\; b_1,\; \cdots,\; b_p,\; a_2,\; \cdots,\; a_m$$

  A fact in the knowledge base is considered as a clause where $p = 0$.

Top-down Proof procedure

- A generalized answer clause is of the form

  $$yes(t_1, \ldots, t_k) \texttt{ :- } a_1,\ a_2,\ \ldots,\ a_m$$

  where $t_1, \ldots, t_k$ are terms and $a_1, \ldots, a_m$ are atoms.
- Select atom in body to resolve against, say a_1.
- The SLD resolution of this generalized answer clause on a_1 with the clause

  $$a \texttt{ :- } b_1,\ \ldots,\ b_p$$

  where a_1 and a have most general unifier θ, is

  $$(yes(t_1, \ldots, t_k) \texttt{ :- } b_1,\ \ldots,\ b_p,\ a_2,\ \ldots,\ a_m)\theta$$

Top-down propositional definite clause interpreter (review)

To solve the query $?q_1, \ldots, q_k$:

> $ac :=$ "yes :- $q_1, \ldots, q_k$"
> **repeat**
> > **select** leftmost atom a_1 from the body of ac
> > **choose** clause C from KB with a_1 as head
> > replace a_1 in the body of ac by the body of C
>
> **until** ac is an answer.

Top-down Proof Procedure

To solve query $?B$ with variables $V_1, \dots, V_k$:

Set *ac* to generalized answer clause $yes(V_1, \dots, V_k)$ `:-` B
while body of *ac* is not empty **do**
  Suppose *ac* is $yes(t_1, \dots, t_k)$ `:-` $a_1,\ a_2,\ \dots,\ a_m$
  select leftmost atom a_1 in the body of *ac*
  choose clause a `:-` $b_1,\ \dots,\ b_p$ in KB
  Rename all variables in a `:-` $b_1,\ \dots,\ b_p$
  Let θ be the most general unifier of a_1 and a.
    Fail if they don't unify
  Set *ac* to $(yes(t_1, \dots, t_k)$ `:-` $b_1,\ \dots,\ b_p,\ a_2,\ \dots,\ a_m)\theta$
end while

Suppose *ac* is generalized answer clause $yes(t_1, \dots, t_k)$ `:-`
Answer is $V_1 = t_1, \dots, V_k = t_k$

Example

$live(Y) \;{:}{-}\; connected\_to(Y, Z),\; live(Z).\;\; live(outside).$

$connected\_to(w_6, w_5).\quad connected\_to(w_5, outside).$

$?live(A).$

$$yes(A) \;{:}{-}\; live(A).$$

$$yes(A) \;{:}{-}\; connected\_to(A, Z_1),\; live(Z_1).$$

$$yes(w_6) \;{:}{-}\; live(w_5).$$

$$yes(w_6) \;{:}{-}\; connected\_to(w_5, Z_2),\; live(Z_2).$$

$$yes(w_6) \;{:}{-}\; live(outside).$$

$$yes(w_6) \;{:}{-}\; .$$

Example

```
elem(E, set(E,_,_)).
elem(V, set(E,LT,_)) :-
    V #< E,
    elem(V,LT).
elem(V, set(E,_,RT)) :-
    E #< V,
    elem(V,RT).
?- elem(3,S),elem(8,S).
```

```
yes(S) :- elem(3,S),elem(8,S)
yes(set(3,S1,S2)) :- elem(8, set(3,S1,S2))
yes(set(3,S1,S2)) :- 3 #< 8, elem(8,S2)
yes(set(3,S1,S2)) :- elem(8,S2)
yes(set(3,S1,set(8,S3,S4))) :-
```

Answer is $S = set(3, S1, set(8, S3, S4))$

Clicker Question

What is the resolution of the generalized answer clause:

$$yes(B, N) \ :-\ append(B, [a, N|R], [b, a, c, d]).$$

with the clause

$$append([], L, L).$$

A $yes([], c) \ :-\ append(B, R, [d])$

B $yes([b], c) \ :-$

C $yes([b|T1], N) \ :-\ append(T1, [a, N|R], [a, c, d]).$

D $yes([b], N) \ :-\ append([], [a, N|R], [a, c, d]).$

E the resolution fails (they do not resolve)

Clicker Question

What is the resolution of the generalized answer clause:

$$yes(B, N) \;:\!-\; append(B, [a, N|R], [b, a, c, d]).$$

with the clause

$$append([H1 \mid T1], A1, [H1 \mid R1]) \;:\!-$$
$$append(T1, A1, R1).$$

- A $yes([], c) \;:\!-\; append(B, R, [d])$
- B $yes([b], c) \;:\!-$
- C $yes([b|T1], N) \;:\!-\; append(T1, [a, N|R], [a, c, d]).$
- D $yes([b], N) \;:\!-\; append([], [a, N|R], [a, c, d]).$
- E the resolution fails (they do not resolve)

Clicker Question

What is the resolution of the generalized answer clause:

$$yes([b|T1], N) \;:\!-\; append(T1, [a, N|R], [a, c, d]).$$

with the clause

$$append([], L, L).$$

A $yes([], c) \;:\!-\; append(B, R, [d])$

B $yes([b], c) \;:\!-$

C $yes([b|T1], N) \;:\!-\; append([], [a, c, d], [a, c, d]).$

D $yes([b], N) \;:\!-\; append([], [a, N|R], [a, c, d]).$

E the resolution fails (they do not resolve)

Unification with function symbols

- Consider a knowledge base consisting of one fact:

  $$lt(X, s(X)).$$

- Should the following query succeed?

  ?- $lt(Y, Y)$.

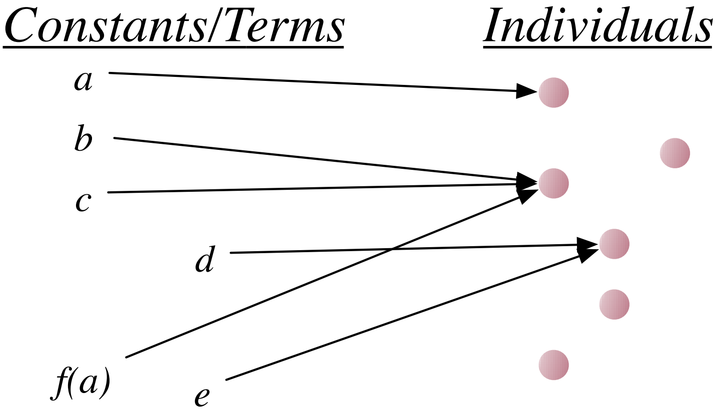
- What does the top-down proof procedure give?
- Solution: variable X should not unify with a term that contains X inside. "Occurs check"
  E.g., X should not unify with $s(X)$.
  Simple modification of the unification algorithm, which Prolog does not do!

Equality

Equality is a special predicate symbol with a standard domain-independent intended interpretation.

- Suppose interpretation $I = \langle D, \phi, \pi \rangle$.
- t_1 and t_2 are ground terms then $t_1 = t_2$ is true in interpretation I if t_1 and t_2 denote the same individual.
  That is, $t_1 = t_2$ if $\phi(t_1)$ is the same as $\phi(t_2)$.
- $t_1 \neq t_2$ when t_1 and t_2 denote different individuals.
- Example:
  $D = \{$✄, ☎, ✎$\}$.
  $\phi(phone) =$ ☎, $\phi(pencil) =$ ✎, $\phi(telephone) =$ ☎
  What equalities and inequalities hold?
  $phone = telephone$, $phone = phone$, $pencil = pencil$,
  $telephone = telephone$
  $pencil \neq phone$, $pencil \neq telephone$
- Equality does not mean similarity!

Equality



Inequality as a subgoal

- What should the following query return?

$$?- \quad X \neq 4.$$

- What should the following query return?

$$?- \quad X \neq 4,\ X = 7.$$

- What should the following query return?

$$?- \quad X \neq 4,\ X = 4.$$

- Prolog has 3 different inequalities that differ on examples like these:

  `\==` `\=` `dif()`

  They differ in cases where there are free variables, and terms unify but are not identical.

3 implementations of not-equals

- Prolog has 3 different inequalities:

  `\==` `\=` `dif()`

  which give same answers for variable-free queries, or when both sides are identical

  `a \== 3,` `a \= 3,` `dif(a,3)`

  all succceed.

  `a \== a,` `a \= a,` `dif(a,a)`

  all fail.

- They give different answers when there is a free variable.
  `\==` means "not identical". `a \== X` succeeds
  `\=` means "not unifiable". `a \= X` fails
  `dif` is less procedural and more logical

Implementing dif

- *dif*(X, Y)
  - all instances fail when X and Y are identical
  - all instances succeed when X and Y do not unify
  - otherwise some instance succeed and some fail
- To implement *dif*(X, Y) in the body of a clause:
  - Select leftmost clause — unless it is a *dif* which cannot be determined to fail or succeed (delay dif calls)
  - Return the *dif* calls not resolved.
- Consider the calls:

```
dif(X,4), X=7.
dif(X,4), X=4.
dif(X,4), dif(X,7).
```

Other predicates, such as #<, work similarly; but can also include more sophisticaled proof techniques (constraint programming).