

Introduction to multi-relational data mining

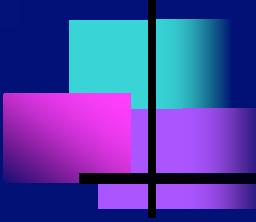


Introduction

Inductive logic programming

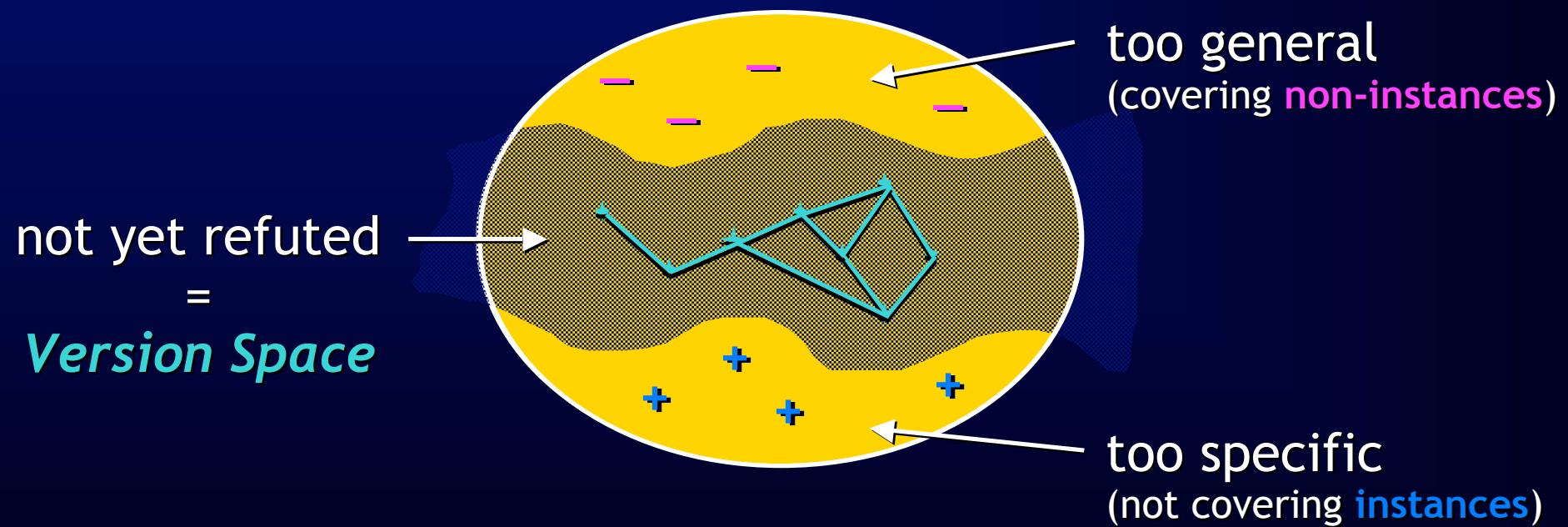
Knowledge representation

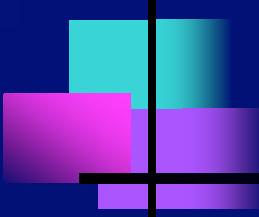
Using other declarative languages



Inductive concept learning

- I **Given:** descriptions of **instances** and **non-instances**
- I **Find:** a **concept covering all instances** and **no non-instances**





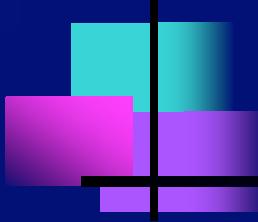
Generalisation and specialisation

- ***Generalising*** a concept involves enlarging its extension in order to cover a given instance or subsume another concept.

- ***Specialising*** a concept involves restricting its extension in order to avoid covering a given instance or subsuming another concept.

- **LGG** = Least General Generalisation

- **MGS** = Most General Specialisation



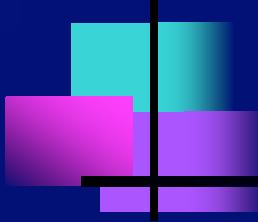
First-order representations

I Propositional representations:

- | datacase is *fixed-size vector of values*
- | features are those given in the dataset

I First-order representations:

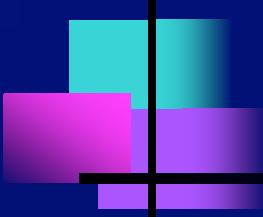
- | datacase is *flexible-size, structured object*
 - | sequence, set, graph
 - | hierarchical: e.g. set of sequences
- | features need to be *selected* from potentially infinite set



Predicting carcinogenicity

■ A molecular compound is carcinogenic if:

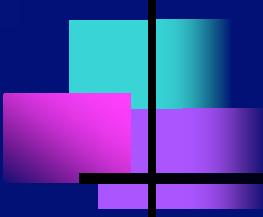
- (1) it tests positive in the Salmonella assay; or
- (2) it tests positive for sex-linked recessive lethal mutation in Drosophila; or
- (3) it tests negative for chromosome aberration; or
- (4) it has a carbon in a six-membered aromatic ring with a partial charge of -0.13; or
- (5) it has a primary amine group and no secondary or tertiary amines; or
- (6) it has an aromatic (or resonant) hydrogen with partial charge ≥ 0.168 ; or
- (7) it has an hydroxy oxygen with a partial charge ≥ -0.616 and an aromatic (or resonant) hydrogen; or
- (8) it has a bromine; or
- (9) it has a tetrahedral carbon with a partial charge ≤ -0.144 and tests positive on Progol's mutagenicity rules.



Concept learning in logic

■ Given:

- *positive examples P*: ground facts to be entailed,
 - *negative examples N*: ground facts not to be entailed,
 - *background theory B*: a set of predicate definitions;
-
- ## ■ Find:
- a *hypothesis H* (one or more predicate definitions) such that
 - for every $p \in P$: $B \cup H \models p$ (*completeness*),
 - for every $n \in N$: $B \cup H \not\models n$ (*consistency*).



Clausal logic

I predicate logic:

$$\forall X: \text{bachelor}(X) \leftrightarrow \text{male}(X) \wedge \text{adult}(X) \wedge \neg \text{married}(X)$$

I clausal logic:

`bachelor(X) ; married(X) :- male(X) , adult(X) .`

`male(X) :- bachelor(X) .`

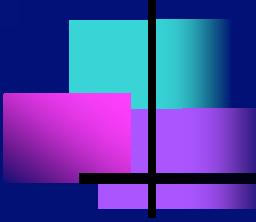
`adult(X) :- bachelor(X) .`

`:- bachelor(X) , married(X) .`

←
indefinite clause

←
definite (Horn) clauses

←
denial



Prolog

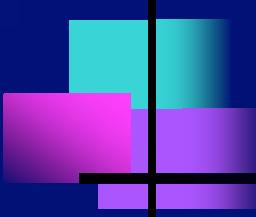
I Ancestors:

- | ancestor(X, Y) :- parent(X, Y) .
- | ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y) .

I Lists:

- | member(X, [X|Z]) .
- | member(X, [Y|Z]) :- member(X, Z) .

- | append([], X, X) .
- | append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs) .



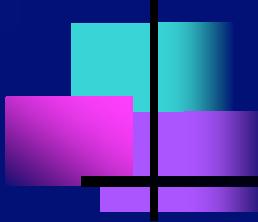
ILP methods

■ bottom-up:

- data-driven approach
- start with long, specific clause
- generalise by applying inverse substitutions and/or removing literals

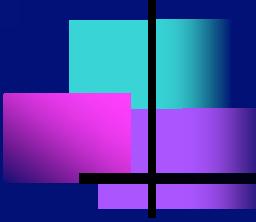
■ top-down:

- generate-then-test approach
- start with short, general clause
- specialise by applying substitutions and/or adding literals



Top-down induction: example

<i>example</i>	<i>action</i>	<i>hypothesis</i>
+ $p(b, [b])$	add clause	$p(X, Y).$
- $p(x, [])$	specialise	$p(X, [V W]).$
- $p(x, [a, b])$	specialise	$p(X, [X W]).$
+ $p(b, [a, b])$	add clause	$p(X, [X W]).$ $p(X, [V W]) :- p(X, W).$



Bottom-up induction: example

- Treat positive examples + ground background facts as **body**
- Choose two examples as **heads** and **anti-unify**

`q([1,2],[3,4],[1,2,3,4]) :-`

`q([1,2],[3,4],[1,2,3,4]), q([a],[],[a]), q([],[],[]), q([2],[3,4],[2,3,4])`

`q([a],[],[a]) :-`

`q([1,2],[3,4],[1,2,3,4]), q([a],[],[a]), q,[],[],[], q([2],[3,4],[2,3,4])`

`q([A|B],C,[A|D]) :-`

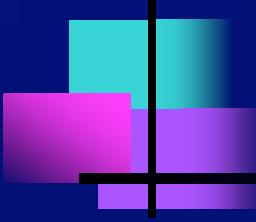
`q([1,2],[3,4],[1,2,3,4]), q([A|B],C,[A|D]), q(W,C,X), q([S|B],[3,4],[S,T,U|V]),`

`q([R|G],K,[R|L]), q([a],[],[a]), q(Q,[],Q), q([P],K,[P|K]),`

`q(N,K,O), q(M,[],M), q,[],[],[], q(G,K,L),`

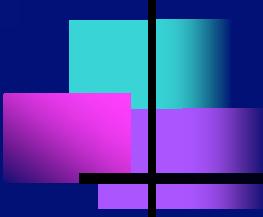
`q([F|G],[3,4],[F,H,I|J]), q([E],C,[E|C]), q(B,C,D), q([2],[3,4],[2,3,4])`

- Generalise by **removing literals** until negative examples covered

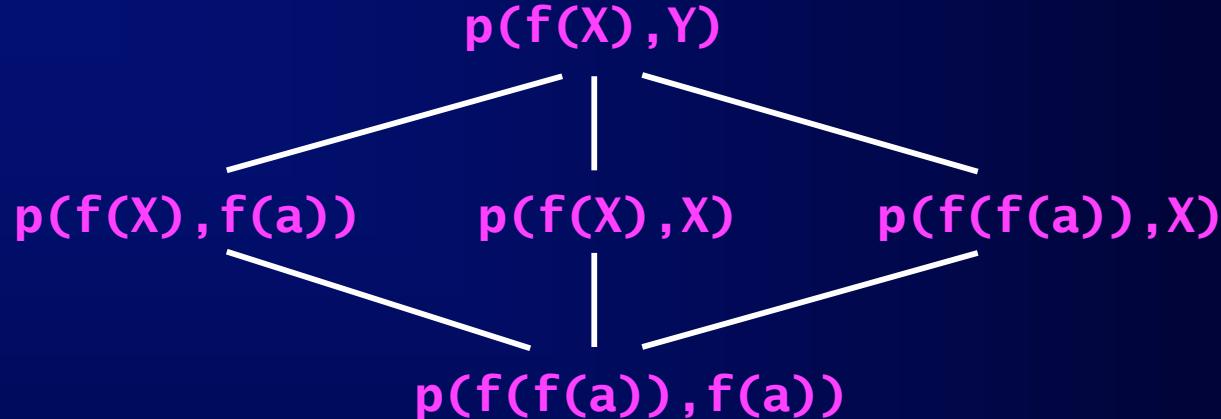


Generality

- | Generality is primarily an **extensional** notion:
 - | one predicate definition is more general than another if its **extension** is a proper **superset** of the latter's extension
- | This can be used to **structure** and **prune** the hypothesis space
 - | if a rule does not cover a positive example, none of its specialisations will
 - | if a rule covers a negative example, all of its generalisations will
- | We need an **intensional** notion of generality, operating on formulae rather than extensions
 - | generality of terms, clauses, and theories

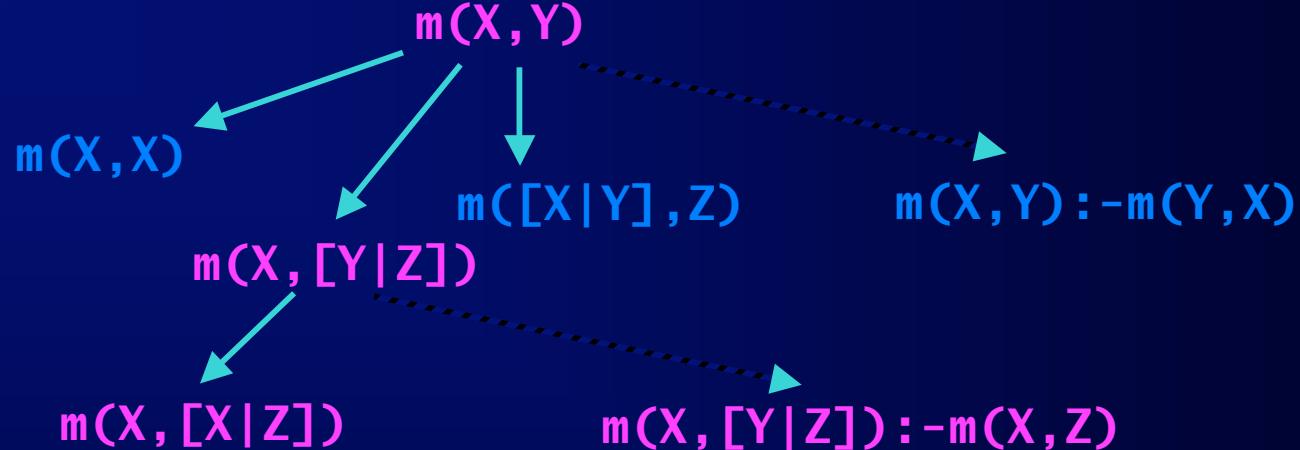


Generality of terms

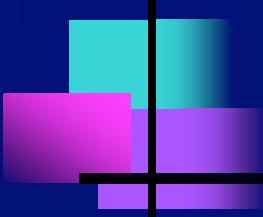


- | The set of first-order terms is a **lattice**:
 - | t_1 is more general than t_2 iff for some substitution θ : $t_1\theta = t_2$
 - | glb \Rightarrow unification, lub \Rightarrow anti-unification
 - | Specialisation \Rightarrow applying a substitution
 - | Generalisation \Rightarrow applying an inverse substitution

Generality of clauses



- | The set of (equivalence classes of) clauses is a **lattice**:
 - | C_1 is more general than C_2 iff for some substitution θ : $C_1\theta \subseteq C_2$
 - | $\text{glb} \Rightarrow \theta\text{-MGS}$, $\text{lub} \Rightarrow \theta\text{-LGG}$
 - | Specialisation \Rightarrow applying a substitution and/or adding a literal
 - | Generalisation \Rightarrow applying an inverse substitution and/or removing a literal
 - | NB. There are infinite chains!



θ -LGG: examples

a([1,2],[3,4],[1,2,3,4]) :- a([2],[3,4],[2,3,4])

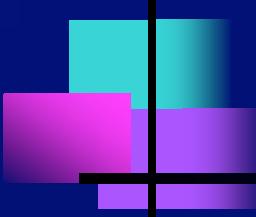
a([a] , [] , [a]) :- a([],[],[])

a([A|B],C , [A|D]) :- a(B,C , D)

m(c,[a,b,c]) :- m(c,[b,c]), m(c,[c])

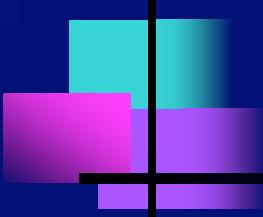
m(a,[a,b]) :- m(a,[a])

m(P,[a,b|Q]) :- m(P,[R|Q]), m(P,[P])



θ -subsumption vs. implication

- Logical implication is **strictly stronger** than θ -subsumption
 - | e.g. $p([V|W]):-p(W) = p([X,Y|Z]):-p(Z)$
 - | this happens when the resolution derivation requires the left-hand clause more than once
- i-LGG of definite clauses is **not unique**
 - | i-LGG($p([A,B|C]):-p(C)$, $p([P,Q,R|S]):-p(S)$) =
 $\{p([X|Y]):-p(Y)$, $p([X,Y|Z]):-p(V)\}$
- Logical implication between clauses is undecidable, θ -subsumption is NP-complete

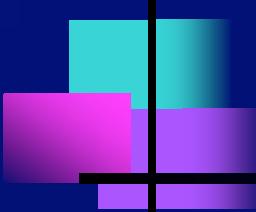


Generality of theories

- Simplification 1: $T_1 = B \cup \{C_1\}$ and $T_2 = B \cup \{C_2\}$ differ just in one clause
- Simplification 2: approximate B by finite ground model B'
- form clauses C_{1B} and C_{2B} by adding ground facts in B' to bodies
- $\theta\text{-RLGG}(C_1, C_2, B) = \theta\text{-LGG}(C_{1B}, C_{2B})$

θ -RLGG: example

```
a([1,2],[3,4],[1,2,3,4]) :-  
    a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),  
    a([],[],[]), a([2],[3,4],[2,3,4]).  
  
a([a] ,[],[a] ) :-  
    a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),  
    a([],[],[]), a([2],[3,4],[2,3,4]).  
  
a([A|B],C , [A|D] ) :-  
    a([1,2],[3,4],[1,2,3,4]), a([A|B],C,[A|D]), a(E,C,F),  
    a([G|B],[3,4],[G,H,I|J]),  
    a([K|L,M,[K|N]), a([a],[],[a]), a(0,[],0),  
    a([P],M,[P|M]),  
    a(Q,M,R), a(S,[],S), a([],[],[]), a(L,M,N),  
    a([T|L],[3,4],[T,U,V|W]), a(X,C,[X|C]), a(B,C,D),  
    a([2],[3,4],[2,3,4]).
```



θ -RLGG: example

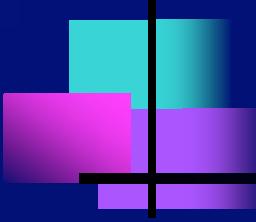
```
a([1,2],[3,4],[1,2,3,4]) :-  
    a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),  
    a([],[],[]), a([2],[3,4],[2,3,4]).
```

```
a([a] ,[],[a] ) :-  
    a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),  
    a([],[],[]), a([2],[3,4],[2,3,4]).
```

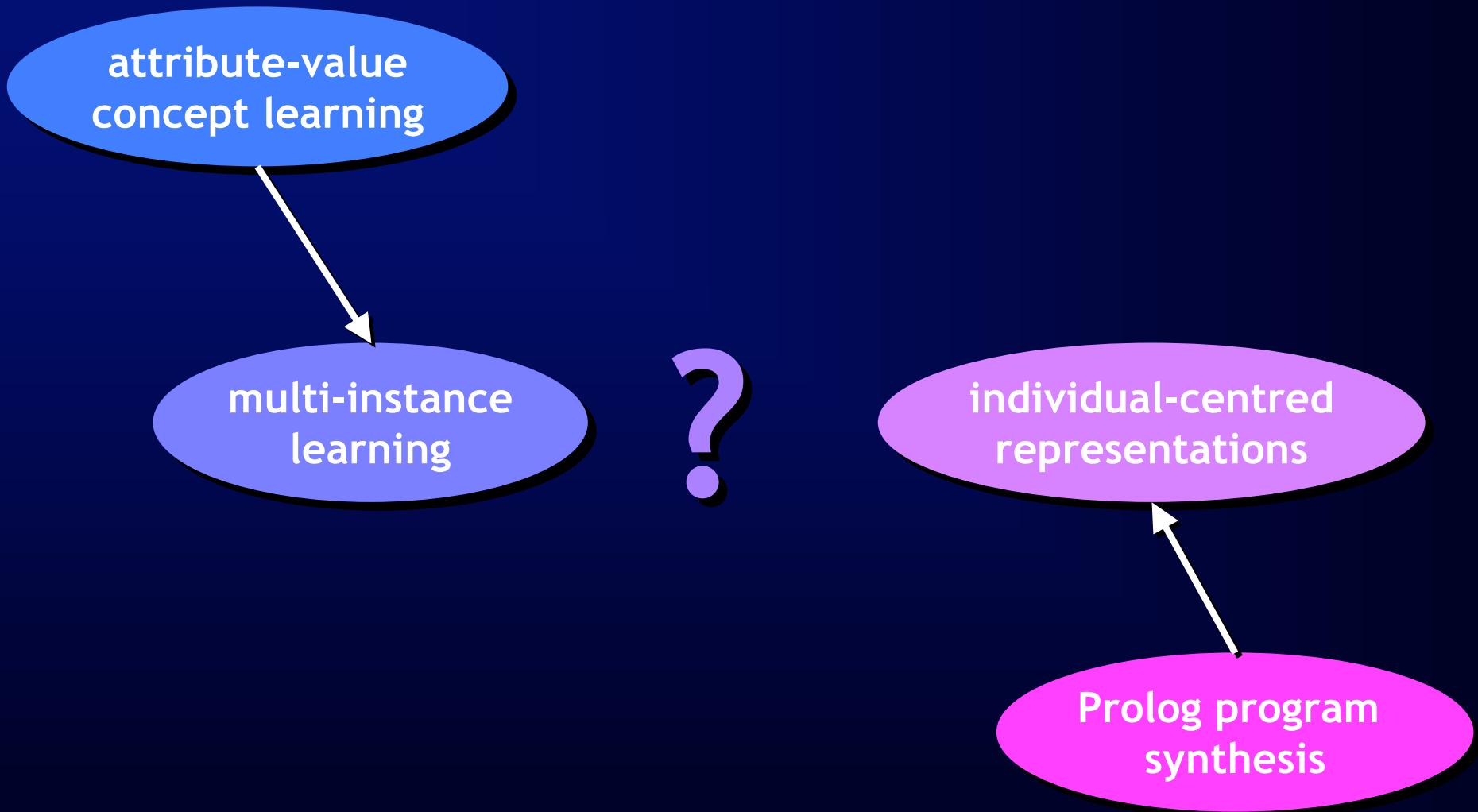
```
a([A|B],C , [A|D] ) :-  
    a([1,2],[3,4],[1,2,3,4]), a([A|B],C,[A|D]), a(E,C,F),  
    a([G|B],[3,4],[G,H,I|J]),  
    a([K|L,M,[K|N]), a([a],[],[a]), a(0,[],0),  
    a([P],M,[P|M]),  
    a(Q,M,R), a(S,[],S), a([],[],[]), a(L,M,N),  
    a([T|L],[3,4],[T,U,V|W]), a(X,C,[X|C]), a(B,C,D),  
    a([2],[3,4],[2,3,4]).
```

θ -RLGG: example

```
a([1,2],[3,4],[1,2,3,4]) :-  
    a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),  
    a([],[],[]), a([2],[3,4],[2,3,4]).  
  
a([a] ,[],[a] ) :-  
    a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),  
    a([],[],[]), a([2],[3,4],[2,3,4]).  
  
a([A|B],C , [A|D] ) :-  
    a([1,2],[3,4],[1,2,3,4]), a([A|B],C,[A|D]), a(E,C,F),  
    a([G|B],[3,4],[G,H,I|J]),  
    a([K|L,M,[K|N]), a([a],[],[a]), a(0,[],0),  
    a([P],M,[P|M]),  
    a(Q,M,R), a(S,[],S), a([],[],[]), a(L,M,N),  
    a([T|L],[3,4],[T,U,V|W]), a(X,C,[X|C]), a(B,C,D),  
    a([2],[3,4],[2,3,4]).
```



Machine learning vs. ILP

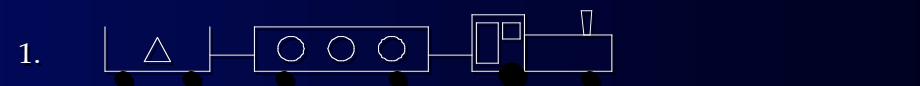
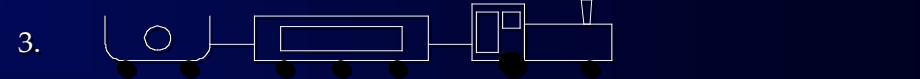
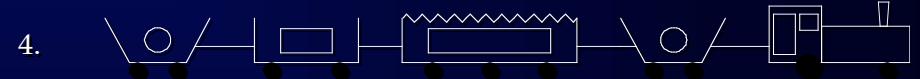
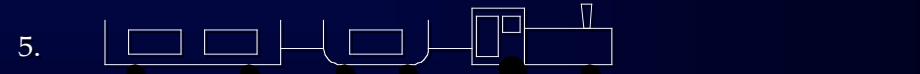


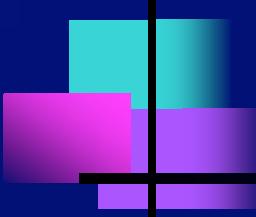
East-West trains

1. TRAINS GOING EAST

1. 
2. 
3. 
4. 
5. 

2. TRAINS GOING WEST

1. 
2. 
3. 
4. 
5. 



East-West trains (flattened)

- Example:
eastbound(t1).

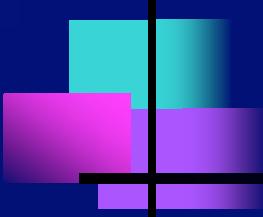


- Background knowledge:

```
hasCar(t1,c1).    hasCar(t1,c2).    hasCar(t1,c3).    hasCar(t1,c4).  
cshape(c1,rect).  cshape(c2,rect).  cshape(c3,rect).  cshape(c4,rect).  
clength(c1,short). clength(c2,long). clength(c3,short). clength(c4,long).  
croof(c1,none).   croof(c2,none).   croof(c3,peak).   croof(c4,none).  
cwheels(c1,2).    cwheels(c2,3).    cwheels(c3,2).    cwheels(c4,2).  
hasLoad(c1,l1).   hasLoad(c2,l2).   hasLoad(c3,l3).   hasLoad(c4,l4).  
lshape(l1,circ).  lshape(l2,hexa).  lshape(l3,tria).  lshape(l4,rect).  
lnumber(l1,1).    lnumber(l2,1).    lnumber(l3,1).    lnumber(l4,3).
```

- Hypothesis:

```
eastbound(T):-hasCar(T,C),clength(C,short),  
not croof(C,none).
```



East-West trains (flattened)

- Example:
eastbound(t1).

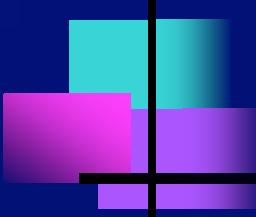


- Background knowledge:

hasCar(t1,c1). hasCar(t1,c2). **hasCar(t1,c3).** hasCar(t1,c4).
cshape(c1,rect). cshape(c2,rect). cshape(c3,rect). cshape(c4,rect).
clength(c1,short). **clength(c2,long).** **clength(c3,short).** **clength(c4,long).**
croof(c1,none). croof(c2,none). **croof(c3,peak).** croof(c4,none).
cwheels(c1,2). cwheels(c2,3). cwheels(c3,2). cwheels(c4,2).
hasLoad(c1,l1). hasLoad(c2,l2). hasLoad(c3,l3). hasLoad(c4,l4).
lshape(l1,circ). lshape(l2,hexa). lshape(l3,tria). lshape(l4,rect).
lnumber(l1,1). lnumber(l2,1). lnumber(l3,1). lnumber(l4,3).

- Hypothesis:

eastbound(T) :- **hasCar(T,C), clength(C,short),**
not croof(C,none).



East-West trains (terms)



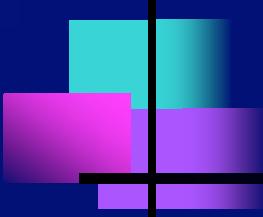
- Example:

```
eastbound([car(rect,short,none,2,load(circ,1)),  
          car(rect,long, none,3,load(hexa,1)),  
          car(rect,short,peak,2,load(tria,1)),  
          car(rect,long, none,2,load(rect,3))]).
```

- Background knowledge: member/2, arg/3

- Hypothesis:

```
eastbound(T):-member(C,T),arg(2,C,short),  
           not arg(3,C,none).
```



East-West trains (terms)



- Example:

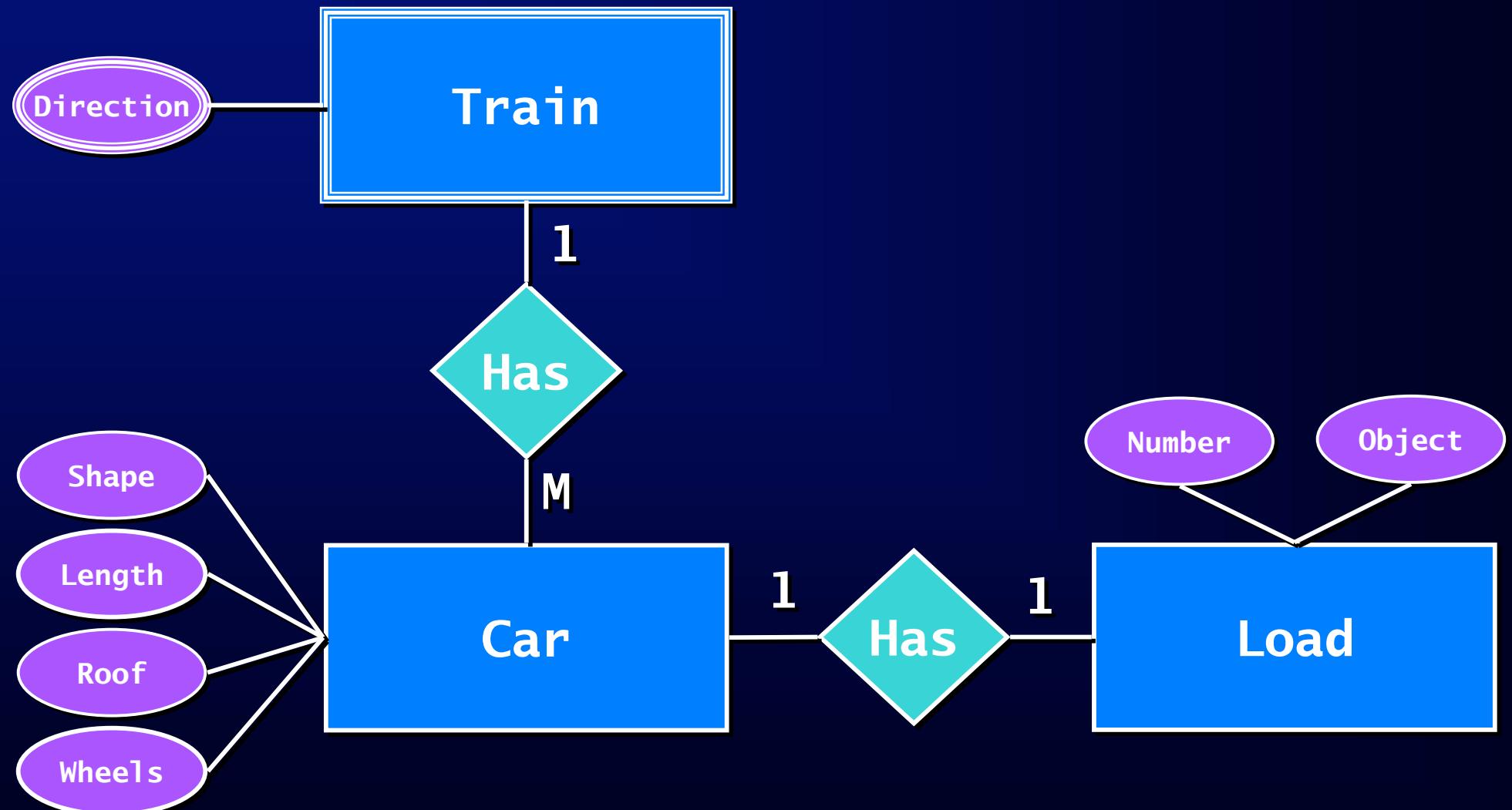
```
eastbound([car(rect,short,none,2,load(circ,1)),  
          car(rect,long, none,3,load(hexa,1)),  
          car(rect,short,peak,2,load(tria,1)),  
          car(rect,long, none,2,load(rect,3))]).
```

- Background knowledge: member/2, arg/3

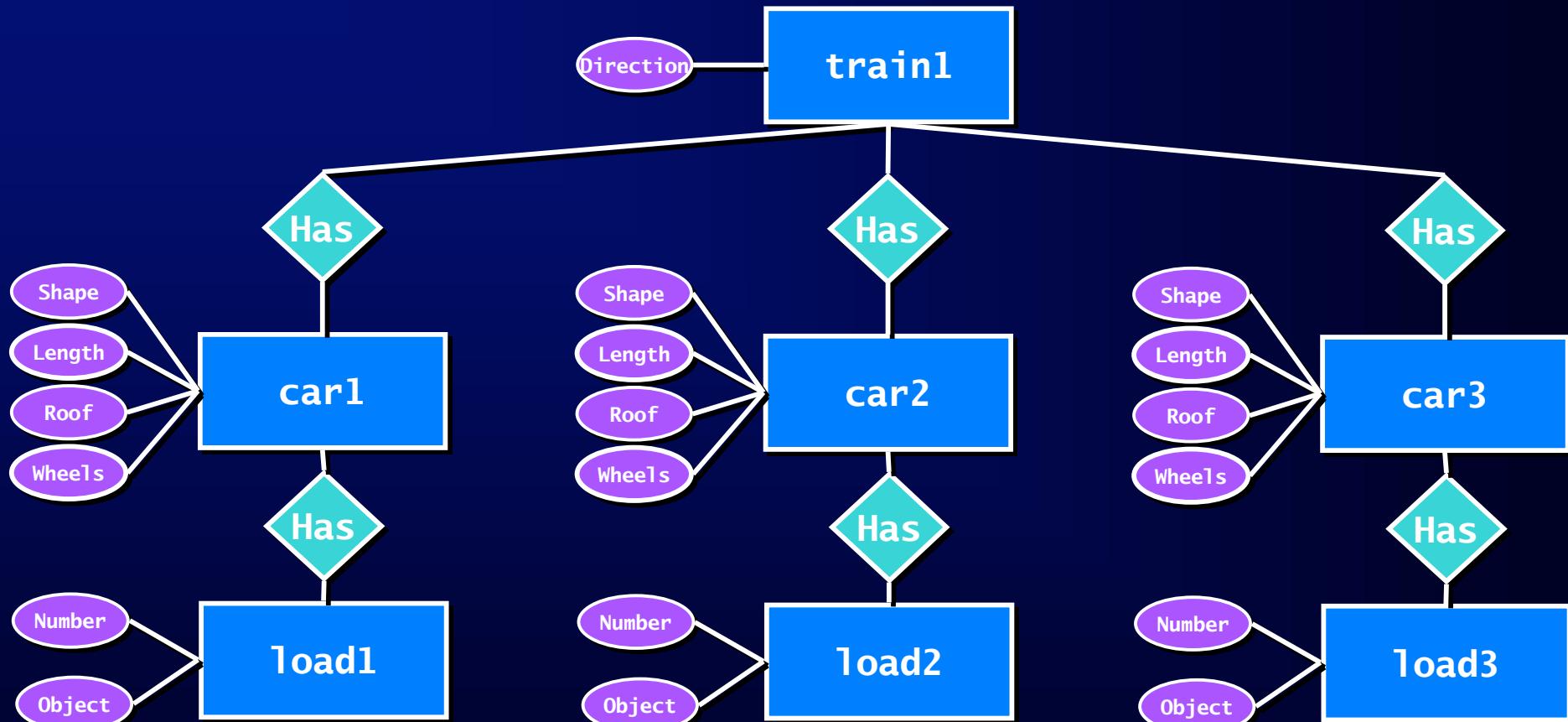
- Hypothesis:

```
eastbound(T):-member(C,T),arg(2,C,short),  
           not arg(3,C,none).
```

ER diagram for East-West trains



Each train is a structured object



Train-as-set database

LOAD_TABLE

<u>LOAD</u>	CAR	OBJECT	NUMBER
I1	c1	circle	1
I2	c2	hexagon	1
I3	c3	triangle	1
I4	c4	rectangle	3
...	

TRAIN_TABLE

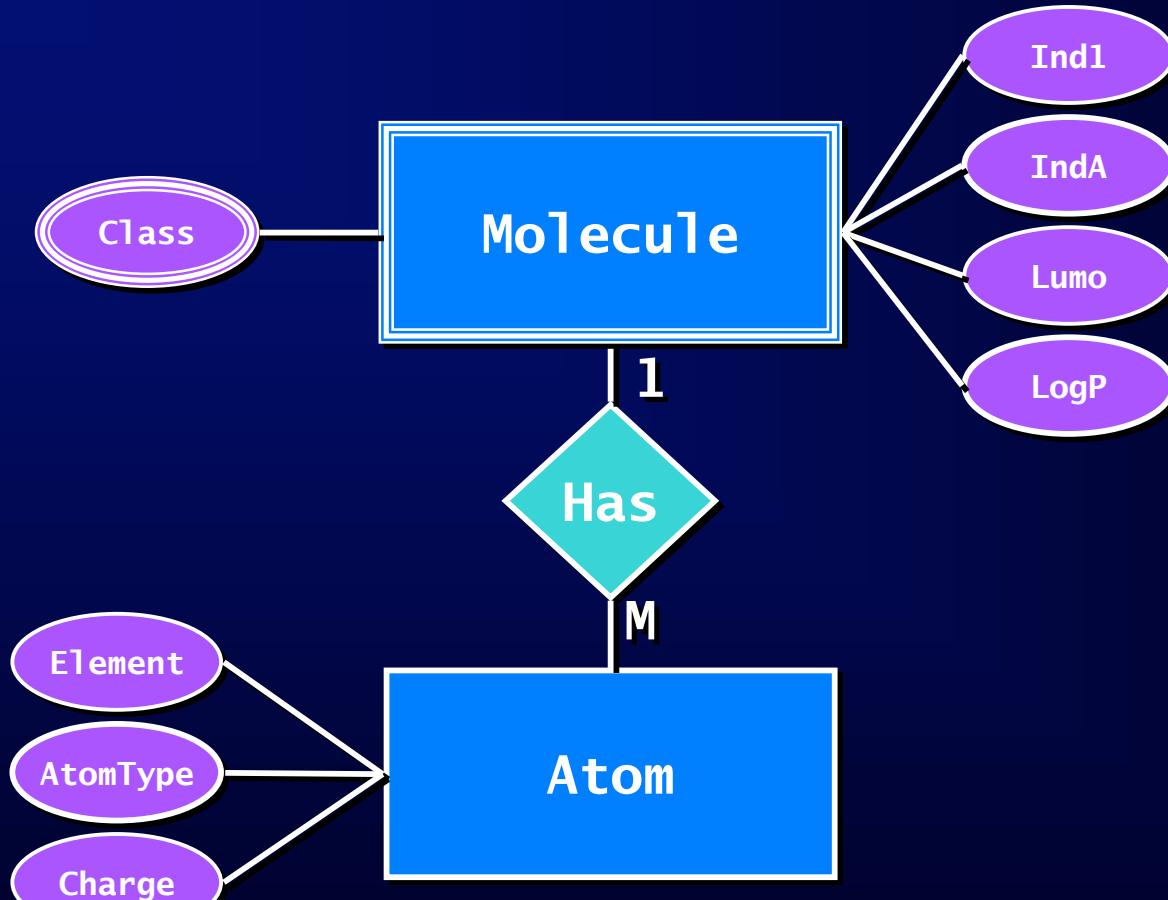
<u>TRAIN</u>	DIRECTION
t1	EAST
t2	EAST
...	...
t6	WEST
...	...

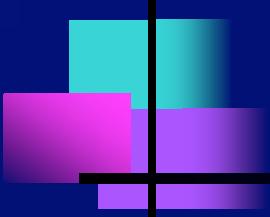
CAR_TABLE

<u>CAR</u>	TRAIN	SHAPE	LENGTH	ROOF	WHEELS
c1	t1	rectangle	short	none	2
c2	t1	rectangle	long	none	3
c3	t1	rectangle	short	peaked	2
c4	t1	rectangle	long	none	2
...

```
SELECT DISTINCT TRAIN_TABLE.TRAIN FROM TRAIN_TABLE, CAR_TABLE WHERE  
TRAIN_TABLE.TRAIN = CAR_TABLE.TRAIN AND  
CAR_TABLE.LENGTH = 'short' AND  
CAR_TABLE.ROOF != 'none'
```

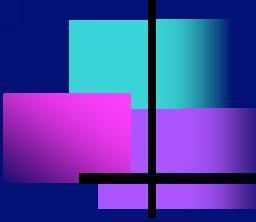
Mutagenesis





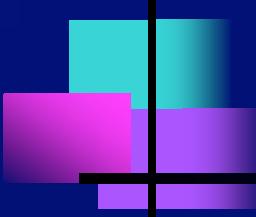
Propositionalising mutagenesis

```
mutagenic(M, false) :- not (has_atom(M, A), atom_type(A, 21)),  
    logP(M, L), L > 1.99, L < 5.64.  
mutagenic(M, false) :- not (has_atom(M, A), atom_type(A, 195)),  
    lumo(M, Lu), Lu > -1.74, Lu < -0.83,  
    logP(M, L), L > 1.81.  
mutagenic(M, false) :- lumo(M, Lu), Lu > -0.77.  
  
mutagenic(M, true) :- has_atom(M, A), atom_type(A, 21),  
    lumo(M, Lu), Lu < -1.21.  
mutagenic(M, true) :- logP(M, L), L > 5.64, L < 6.36.  
mutagenic(M, true) :- lumo(M, Lu), Lu > -0.95,  
    logP(M, L), L < 2.21.
```



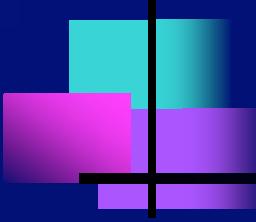
Individual-centred representations

- ER diagram is a tree (approximately)
 - root denotes individual
 - looking downwards from the root, only one-to-one or one-to-many relations are allowed
 - one-to-one cycles are allowed
- Database can be partitioned according to individual
- Alternative: all information about a single individual packed together in a term
 - tuples, lists, sets, multisets, trees, ...



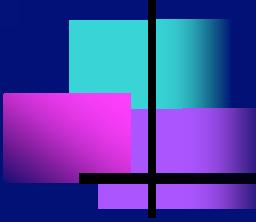
Complexity of ILP problems

- Simplest case: single table with primary key
 - example corresponds to tuple of constants
 - *attribute-value* or *propositional* learning
- Next: single table without primary key
 - example corresponds to set of tuples of constants
 - *multiple-instance* problem
- Complexity resides in many-to-one foreign keys
 - lists, sets, multisets
 - *non-determinate* variables



Example: Escher

- higher-order functional logic programming
 - real sets, real functions
- strongly typed
- inherits Haskell syntax
 - variables start with lowercase, constants start with uppercase



Attribute-value learning in Escher

■ Type definitions:

```
playTennis::Weather->Bool;  
type Weather = (Outlook, Temperature, Humidity, Wind)  
data Outlook = Sunny | Overcast | Rain;  
data Temperature = Hot | Mild | Cool;  
data Humidity = High | Normal | Low;  
data Wind = Strong | Medium | Weak;
```

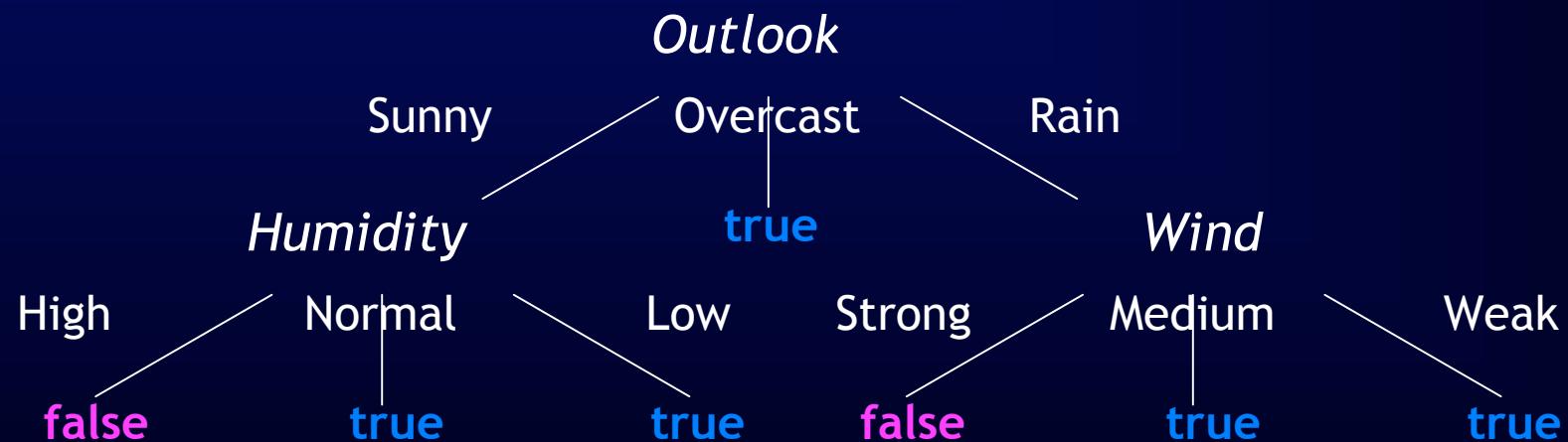
■ Examples:

```
playTennis(Overcast, Hot, High, Weak) = True;  
playTennis(Sunny, Hot, High, Weak) = False;
```

Attribute-value learning in Escher

Hypothesis:

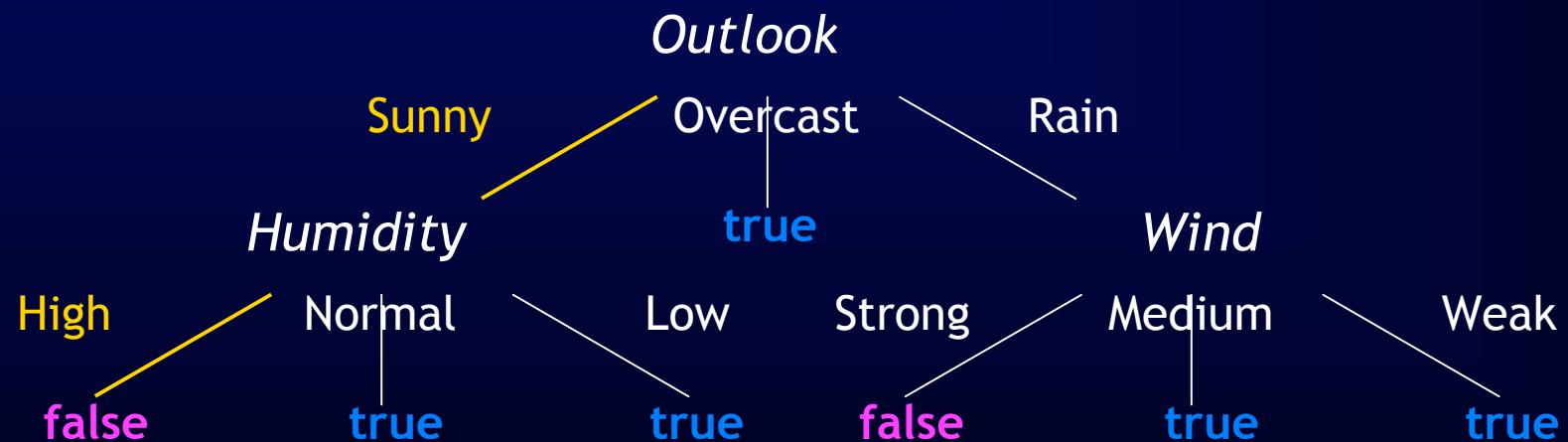
```
outlookP::Weather->Outlook;  
outlookP(o,t,h,w) = o;  
  
playTennis(w) =  
  if (outlookP(w)==Sunny && humidityP(w)==High) then False  
  else if (outlookP(w)==Rain && windP(w)==Strong) then False  
  else True;
```



Attribute-value learning in Escher

Hypothesis:

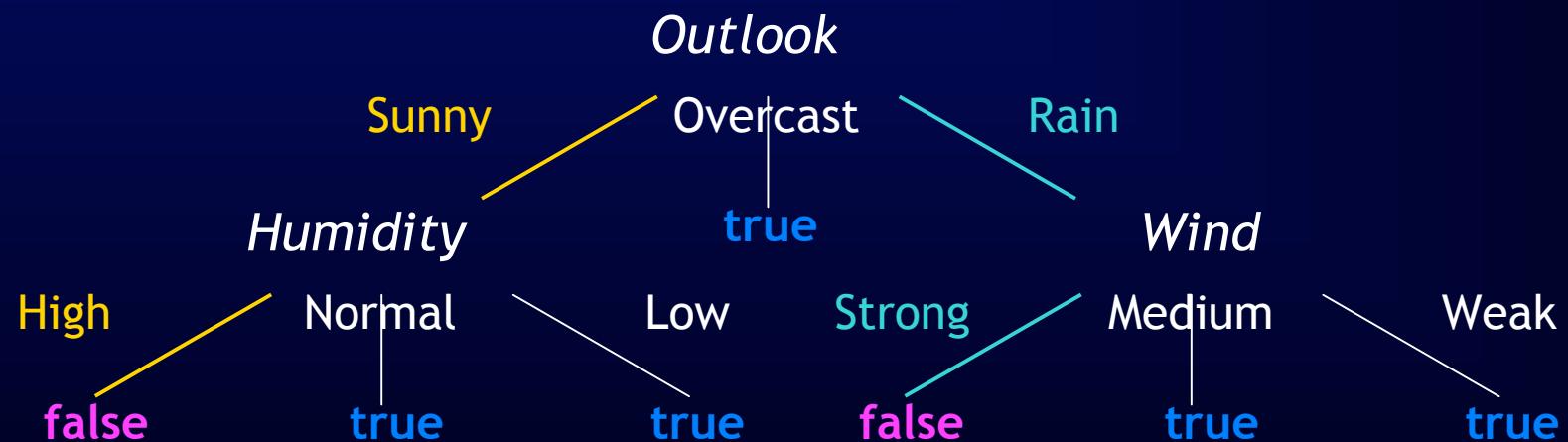
```
playTennis(w) =  
  if (outlookP(w)==Sunny && humidityP(w)==High) then False  
  else if (outlookP(w)==Rain && windP(w)==Strong) then False  
  else True;
```

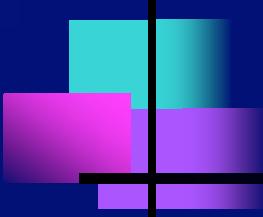


Attribute-value learning in Escher

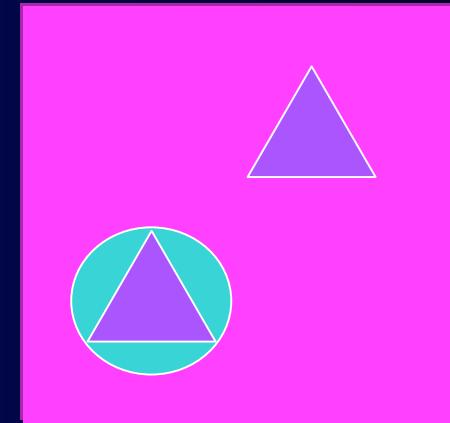
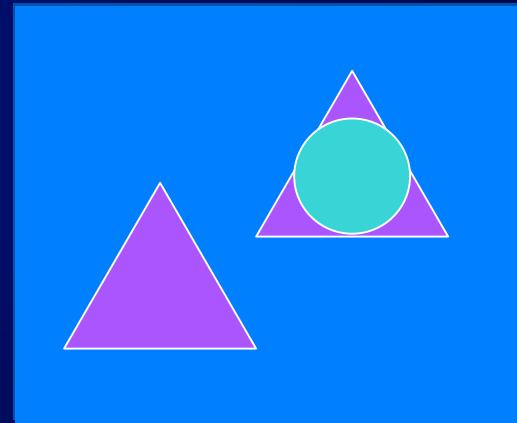
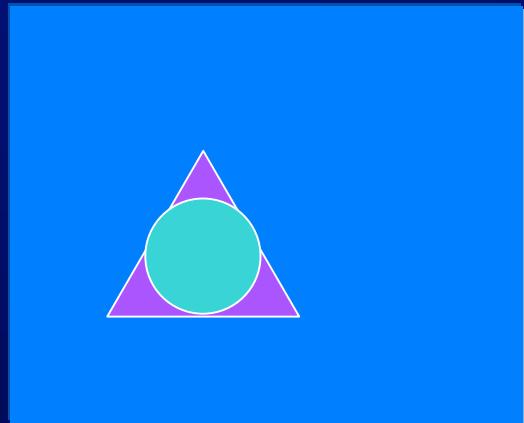
Hypothesis:

```
playTennis(w) =  
    if (outlookP(w)==Sunny && humidityP(w)==High) then False  
    else if (outlookP(w)==Rain && windP(w)==Strong) then False  
    else True;
```



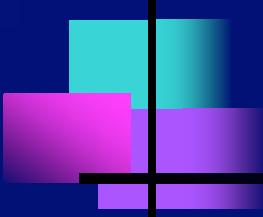


Multi-instance learning in Escher



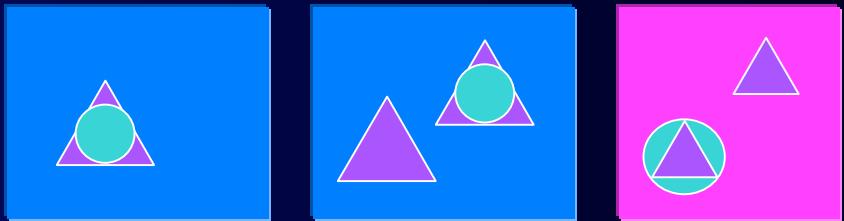
I Type definitions:

```
class::Diagram->Class;  
type Diagram = {(Shape,Int)};  
data Shape = Circle | Triangle | In(Shape,Shape);  
data Class = Positive | Negative;
```



Multi-instance learning in Escher

■ Examples:



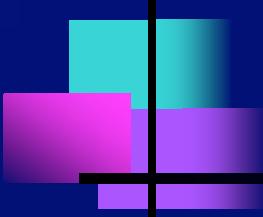
```
class({(In(Circle, Triangle), 1)}) = Positive;
```

```
class({(Triangle, 1), (In(Circle, Triangle), 1)}) = Positive;
```

```
class({(In(Triangle, Circle), 1), (Triangle, 1)}) = Negative;
```

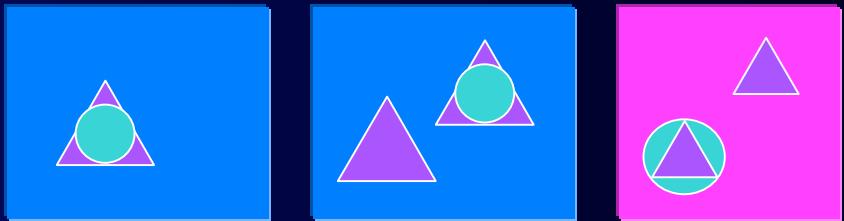
■ Hypothesis:

```
class(d) =  
  if (exists \p -> p 'in' d && (exists \s t ->  
    shapeP(p) == In(s, t) && s == Circle))  
  then Positive else Negative;
```



Multi-instance learning in Escher

■ Examples:



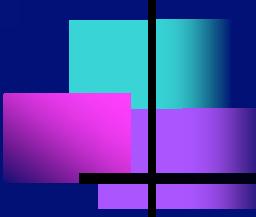
```
class({(In(Circle, Triangle), 1)}) = Positive;
```

```
class({(Triangle, 1), (In(Circle, Triangle), 1)}) = Positive;
```

```
class({(In(Triangle, Circle), 1), (Triangle, 1)}) = Negative;
```

■ Hypothesis:

```
class(d) =  
  if (exists \p -> p 'in' d && (exists \s t ->  
    shapeP(p) == In(s, t) && s == Circle))  
  then Positive else Negative;
```



East-West trains in Escher

I Type signature:

```
eastbound :: Train -> Bool;
```



```
type Train = [Car]; type Car = (CShape, CLength, CRoof, CWheels, Load);  
type Load = (LShape, LNumber);
```

```
data CShape = Rect | Hexa | ...; data CLength = Long | Short;  
data CRoof = None | Peak | ...; data LShape = Circ | Hexa | ...;
```

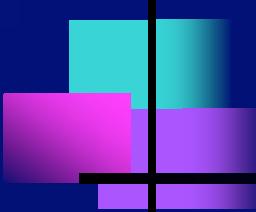
```
type CWheels = Int; type LNumber = Int
```

I Example:

```
eastbound([(Rect, Short, None, 2, (Circ, 1)),  
           (Rect, Long, None, 3, (Hexa, 1)),  
           (Rect, Short, Peak, 2, (Tria, 1)),  
           (Rect, Long, None, 2, (Rect, 3))]) = True
```

I Hypothesis:

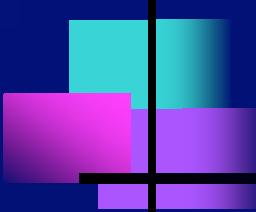
```
eastbound(t) = (exists \c -> member(c, t) &&  
                  CLengthP(c)==Short && CRoofP(c) !=None)
```



Mutagenesis in Escher

I Type signature:

```
mutagenic::Molecule->Bool;  
  
type Molecule = (Ind1, IndA, Lumo, LogP, AtomSet, BondSet);  
type AtomSet = {Atom};  
type Atom = (AtomID, Element, AtomType, Charge);  
type BondSet = {Bond};  
type Bond = (AtomIDSet, BondType);  
type AtomIDSet = {AtomID}  
  
type Ind1 = Bool;           type AtomID = Int;  
type IndA = Bool;           type AtomType = Int;  
type Lumo = Float;          type Charge = Float;  
type LogP = Float;          type BondType = Int;  
  
data Element = Br | C | Cl | F | H | I | N | O | S;
```



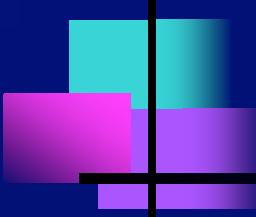
Mutagenesis in Escher

■ Examples:

```
mutagenic(True, False, -1.246, 4.23,
  { (1,C,22,-0.117),
    (2,C,22,-0.117),
    ...,
    (26,O,40,-0.388) },
  { ({1,2},7),
    ...,
    ({24,26},2) })
= True;
```

} atoms
} bonds

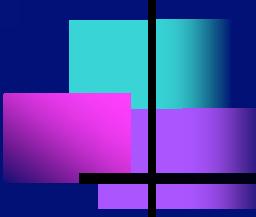
■ **Naming** of sub-terms cannot be avoided here, because molecules are graphs rather than trees



Mutagenesis in Escher

Hypothesis:

```
mutagenic(m) =  
  ind1P(m) == True || lumoP(m) <= -2.072 ||  
  (exists \a -> a 'in' atomSetP(m) && elementP(a)==C &&  
   atomTypeP(a)==26 && chargeP(a)==0.115) ||  
  (exists \b1 b2 -> b1 'in' bondSetP(m) && b2 'in' bondSetP(m) &&  
   bondTypeP(b1)==1 && bondTypeP(b2)==2 &&  
   not disjoint(atomIDSetP(b1),atomIDSetP(b2))) ||  
  (exists \a -> a 'in' atomSetP(m) &&  
   elementP(a)==C && atomTypeP(a)==29 &&  
   (exists \b1 b2 ->  
     b1 'in' bondSetP(m) && b2 'in' bondSetP(m) &&  
     bondTypeP(b1)==7 && bondTypeP(b2)==1 &&  
     atomIDP(a) 'in' atomIDSetP(b1) &&  
     not disjoint(atomIDSetP(b1),atomIDSetP(b2)))) ||  
...;
```



Outlook

- Full program synthesis still much too hard
- Much to be gained from careful representation engineering
- Not restricted to classification rule learning
 - naïve Bayes, instance-based, association rules, knowledge discovery
 - key issue is input format rather than output format