Artificial Intelligence
Lab: Inheritance

1 Introduction

According to our text, Computational Intelligence (page 189), an arc from c to n labeled p, c \rightarrow n, is a representation of the clause

\[ \text{prop(Obj, p, n)} :\! - \! \text{prop(Obj, is-a, c)}. \]

Passing from constants p, n and c to variables Att, Val, and Class (respectively), this principle of inheritance generalizes to

\[ \text{prop(Obj, Att, Val)} :\! - \! \text{prop(Obj, is-a, Class), prop(Class, Att, Val)}. \]

The intuition behind (1) is that an object Obj that is a (member of) Class “inherits” all attribute-values Att, Val of Class.

2 Guarding against loops

Assuming that \( O \Rightarrow C \) means \( O \text{ is-a C} \), the semantic net below says that Fido is a dog, and a dog has 4 legs.

\[ \text{prop(fido, is-a, dog)} \]

Let us form a knowledge base consisting of the two \text{prop} facts above and our principle (1) of inheritance. Against this knowledge base, try the queries

\[ \text{?- prop(fido, legs, 4).} \]

\[ \text{?- prop(fido, legs, 10).} \]

\[ \text{?- prop(fido, is-a, cat).} \]

Trace the last two queries, observing the similarity between (1) and the rule

\[ \text{ancestor(X, Y)} :\! - \! \text{ancestor(X, Z), ancestor(Z, Y)}. \]

To break looping on \text{ancestor} queries that ought to fail, the recursive rule for \text{ancestor} is typically formulated as

\[ \text{ancestor(X, Y)} :\! - \! \text{parent(X, Z), ancestor(Z, Y)}. \]

Similarly, to avoid matching the head \text{prop(Obj, Att, Val)} of (1) with its first assumption \text{prop(Obj, is-a, Class)}, let us rewrite is-a facts without prop, reformulating \text{prop(fido, is-a, dog)} as \text{is-a(fido, dog)}, and (1) as

\[ \text{prop(Obj, Att, Val)} :\! - \! \text{is-a(Obj, Class), prop(Class, Att, Val)}. \]

Re-try the queries above, rephrasing the last as \text{is-a(fido, cat)}. 
3 Inheritance along is-a

Next, let us add to our knowledge base the fact that a dog is a pet.

```
  is-a(dog,pet).
  prop(dog,legs,4).
  is-a(fido,dog).
```

Try the query

```
| ?- is-a(fido,pet).
```

To apply the principle of inheritance to \( \text{Att} = \text{is-a} \), let us break the loop from (1) in another way, splitting \( \text{prop} \) off between two predicates \( \text{prim} \) and \( \text{der} \).

We use \( \text{prim} \) to record explicitly given facts such as

```
  prim(dog,is-a,pet).
  prim(dog,legs,4).
  prim(fido,is-a,dog).
```

We reserve \( \text{der} \) for derived facts, reformulating (2) as

```
  der(OC,Att,Val) :- prim(OC,Att,Val).
```

```
  % (3)
  der(OC,Att,Val) :- prim(OC,is-a,Class), der(Class,Att,Val).
```

Try the queries

```
| ?- prim(fido,is-a,pet).
| ?- der(fido,is-a,pet).
```

4 Exceptions over-riding defaults

Now, another dog, Rover, comes along, and bites off one of Fido’s legs.

```
  is-a(rover,pet).
  prop(rover,legs,4).
  is-a(fido,dog).
```

Try the queries

```
| ?- der(rover,legs,X).
| ?- der(fido,legs,3).
| ?- der(fido,legs,4).
```

To block the derivation that Fido has 4 legs, change (3) to
der(OC,Att,Val) :-
    \+ prim(OC,Att,_,) ; multiValued(Att),
                   % not already specified
    prim(OC,is-a,Class), der(Class,Att,Val).

% \+ is a built-in Prolog function for negation-as-failure
% negation-as-failure(P) :- P,! fail ; true.

5 Multi-valued attributes

But now re-try the query

?- der(fido,is-a,pet).

Unlike legs, is-a is a multi-valued attribute. Accordingly, change (4) to

der(OC,Att,Val) :-
    \+ prim(OC,Att,_) ; multiValued(Att),
                   % not already specified
    prim(OC,is-a,Class), der(Class,Att,Val).

multiValued(is-a).

Re-try the queries above.

6 Favoring specificity

Suppose we agreed that pets have 6 legs (e.g. spiders).

How many legs does Rover have? (5) implements the principle “most specific rule
wins” where specificity falls with every appeal to an is-a arc. What happens if we
were to add an is-a arc from rover to pet? How many legs does Rover get?

7 Addendum

Revise

der(OC,Att,Val) :- prim(OC,Att,Val).

% (4)
der(OC,Att,Val) :-
    \+ prim(OC,Att,_,),
                   % not already specified
    prim(OC,is-a,Class), der(Class,Att,Val).
der(OC, Att, Val) :- prim(OC, Att, X), !, X = Val.
% instead of just
% prim(OC, Att, X), !.
% which fouls up on: der(fido, legs, 4)
der(OC, Att, Val) :-
  prim(OC, is-a, Class), der(Class, Att, Val).

Idea. Move negation-as-failure from recursive clause of der to a cut in the base case of der, avoiding the recomputation of prim(OC, Att, Val).

And for multi-valued predicates, try changing base clause to

der(OC, Att, Val) :- prim(OC, Att, Val),
  (multiValued(Att); !).

4