Week 9

Brendon Faleiro

Prolog

- Recap:
  - Prolog is a logical (declarative) language.
  - Works with “What” and not “How”.
  - You specify logical constraints and the language finds a solution that satisfies those constraints.
  - Your knowledge-base consists of facts/rules.
  - You send queries to the interpreter.
  - The language builds answers that satisfy these rules using the facts.

Variables and Syntax

- Prolog variables begin with a capital letter.

- Prolog Term Syntax:

t ::= c | X | c(t1,...,tn)

c ::= constant (lowercase identifier)

X ::= variable (uppercase identifier)

Rules and Syntax

- AND

```
p(X) :- a(X) , b(X) .                                   /*if a(X) and b(X) , then p(x) */
mother(X, Y) :- parent(X, Y), female(X).
father(X, Y) :- parent(X, Y), male(X).
```

- OR

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
/* if mother(X, Y) then parent(X, Y) or if father(X, Y) then parent(X, Y) */
```

- These rules can be recursive.

Prolog Queries

- Queries are questions you ask the interpreter.

- These can either be Yes/No questions or Wh- questions.

- To ask Yes/No questions:
  - No variables.
  - E.g. cat(tom). /* is tom a cat? */
  - E.g. sister(kylie, kendall). /* are kylie and kendall sisters? */

- To ask Wh- questions:
  - Use variables.
  - E.g. cat(X). /* Who is a cat? */
  - E.g. sisters(X, bella). /* Who is bella's sister? */

Unification

- All the power in the language comes from Prolog's unification.
- Unification takes two terms t1 and t2 and returns either NO or an environment (mapping variables to terms) that makes t1 and t2 syntactically identical.
- Unification in prolog helps it in solving problems in applications like type inferencing.

Unification Rules

- c = c ---> {}
- X = t ---> {X:t}
- t = X ---> {X:t}
- c(t1,t2) = c(t1',t2') --->

```
result1 = (t1=t1')
if (result1 == NO)
        return NO
else
        result2 = (result1(t2) = result1(t2’))
        if (result2 == NO)
                return NO
        else
                return result1 U result2
```

- otherwise NO

Prolog Notation	Unifying Substitution	Explanation
a = a	{}	Succeeds. (tautology)
a = b	⊥	*a* and *b* do not match
X = X	{}	Succeeds. (tautology)
a = X	{ *x* ↦ *a* }	*x* is unified with the constant *a*
X = Y	{ *x* ↦ *y* }	*x* and *y* are aliased
f(a,X) = f(a,b)	{ *x* ↦ *b* }	function and constant symbols match, *x* is unified with the constant *b*
f(a) = g(a)	⊥	*f* and *g* do not match
f(X) = f(Y)	{ *x* ↦ *y* }	*x* and *y* are aliased
f(g(X)) = f(Y)	{ *y* ↦ *g*(*x*) }	Unifies *y* with the term g(x)
f(g(X),X) = f(Y,a)	{ *x* ↦ *a*, *y* ↦ *g*(*a*) }	Unifies *x* with constant *a*, and *y* with the term g(a)
X = f(X)	should be ⊥	Returns ⊥ in first-order logic and many modern Prolog dialects. Succeeds in traditional Prolog and in Prolog II, unifying *x* with infinite term x=f(f(f(f(...)))).
X = Y, Y = a	{ *x* ↦ *a*, *y* ↦ *a* }	Both *x* and *y* are unified with the constant *a*
a = Y, X = Y	{ *x* ↦ *a*, *y* ↦ *a* }	As above (order of equations in set doesn't matter)
X = a, b = X	⊥	Fails. *a* and *b* do not match, so *x* can't be unified with both

Source: https://en.wikipedia.org/wiki/Unification_(computer_science)

Prolog findall

- Prolog typically finds one unification result at a time and returns the answer, then asking if you want the system to continue searching.

- findall(Template, Query, Var) produces a list of all solutions to Query in Var, formatted according to Template.

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z),
                descend(Z,Y).
```

```
?- descend(martha,X).
X=charlotte ? ;
X=caroline ? ;
X=laura ? ;
X=rose.

?- findall(X,descend(martha,X),Z).
Z = [charlotte,caroline,laura,rose]

?- findall(fromMartha(X),descend(martha,X),Z).
Z = [fromMartha(charlotte),fromMartha(caroline),
fromMartha(laura),fromMartha(rose)]

?- findall(X,descend(mary,X),Z).
Z= []

?- descend(mary,X).
no
```

?-compatible(john,X).

Knowledge-base:

```
compatible(X,Y) :- reading(X), reading(Y).
compatible(X,Y) :- football(X), football(Y).
compatible(X,Y) :- friends(X,Y).
compatible(X,Y) :- mutual(X,Y).
friends(X,Y) :- havemet(X,Y), compatible(X,Y).
havemet(X,Y) :- met(X,Y).
havemet(X,Y) :- met(Y,X).
mutual(X,Y) :- friends(X,Temp), friends(Y,Temp).
mutual(X,Y) :- friends(Temp,X), friends(Y,Temp).
mutual(X,Y) :- friends(X,Temp), friends(Temp,Y).
mutual(X,Y) :- friends(Temp,X), friends(Temp,Y).
```

```
football(john).
football(james).
friends(john, carl).
friends(carl, john).
reading(carl).
reading(fred).
reading(emily).
met(carl, emily).
met(fred, james).
met(fred, emily).
```