COM S-342

Recitation 11/05/18 – 11/07/18

Today

- Logic Programming Concepts: set, relation, predicate, quantifier, function, backtracking, induction, deduction
- Reflang programming

Logic Programming

- Declarative languages
  - Consist of declarations rather than assignments
- Non-procedural programming languages
  - HOW instead of WHAT
- Programmer defines:
  - Sets of objects
  - Relationships between objects
  - Constraints that need to hold
- Interpreter or compiler
  - Solve equations (how to satisfy constraints)

Logic Programming

- Based on formal logic
- A proposition can be thought of as a logical statement that may or may not be true
  - consists of **objects** and the **relationships** among objects
- Formal logic developed to describe propositions check their validity
- Symbolic logic used to inferred other propositions from true propositions

Logic Programming

- **Individuals** or **objects** = **constants or terms**
- **Relations** over individuals = **properties** or **predicates**
  - They can have arity (number of individuals)
- **Quantifiers**
  - Use to describe **all individuals**, or
  - **Some individuals**

Fact Statements

- Used to construct the hypotheses, or database of information
  - father(bill, jake).
  - father(bill, shelley).
  - mother(mary, jake).
  - mother(mary, shelley).

Rules Statements

- A conclusion can be drawn if a set of given conditions is satisfied
- Right side is called *antecedent* and left side is the *consequence*
- Antecedent can be a single term or a conjuction
- Example:
  - parent(X, Y) :- mother(X, Y).
  - parent(X, Y) :- father(X, Y).
  - grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).

Goals or Queries Statements

- Basis of theorem-proving model
- The theorem is a proposition that we want the system to prove
- Syntax is identical to facts statements
- Example:
  - man(fred). // answer is yes or no
  - father(X, mike). // search for result that satisfies the value of X and results true

Prolog Example

Facts

```
isamother(mary).          %% Horn Clause with no Antecedent
childof(tom, mary).       %% Horn Clause with no Antecedent
```

Rules

```
%% Rule: Horn Clause with antecedent
loves(mary, tom) :-
        isamother(mary), childof(tom, mary).
```

Query

```
%% Query: Horn Clause with no consequent
?- loves(mary, tom).
```

Horn Clauses

- Horn Clause with no Antecedent = **Fact**
- Horn Clause with Antecedent = **Rule**
- Horn Clause with no Consequence = **Query**

- **Logic Programming is a collection of Horn Clauses**

$$c \leftarrow h_1 \vee h_2 \longrightarrow \begin{array}{l} c \leftarrow h_1 \\ c \leftarrow h_2 \end{array}$$

Horn Clauses

Facts

```
isamother(mary).          %% Horn Clause with no Antecedent
childof(tom, mary).       %% Horn Clause with no Antecedent
childof(jerry, mary).     %% Horn Clause with no Antecedent
```

Rules

```
%% Rule: Horn Clause with antecedent and with variables
%%        X and Y are universally quantified
loves(X, Y) :-
        isamother(X), childof(Y, X).

%%        X is universally quantified
%%        Y, Z are existentially quantified
hassibling(X) :-
        childof(X, Y), childof(Z, Y).
```

Query

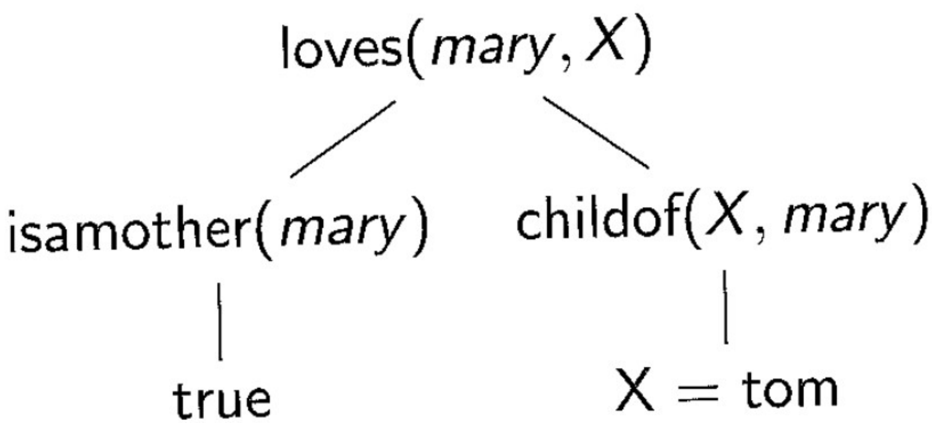
```
%% Query: Horn Clause with no consequent
?- loves(mary, X). %% X is exitentially quantified
?- hassibling(jerry).
```

Queries

Queries:

```
?- loves(mary, X).
```

means: does there exists an X such that loves($mary, X$) is true.



Executing Logic Programs

- Queries are called **goals**
- When a query is a rule, then each antecedent is a **subgoal**
- To prove a a goal is true, the inference must find a chain of inference rules or facts

Executing Logic Programs

- Unification → Variable Binding
- Backward Chaining → Reducing goals into simpler subgoals
- Backtracking → Search for answers

Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?
  yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?
  yes by replacing `X` and `Y` by the same individual
- Unify `childof(jane, X)` and `childof(Y, mary)`?
  yes by replacing `X` by `mary`, and `Y` by `jane`
- Unify `childof(jane, X)` and `childof(tom, Y)`? No.

Computing with Logic

- Given a query
  - Search for facts and rules and,
  - Verify whether the query unifies with any consequence
  - If search fails, return false
  - If search is successful, then
    - if the unification occurs with the consequent of a fact, return the substitution of the variables (if any)
    - if the unification occurs with the consequent of a rule, instantiate the variables (if any) and prove the subgoals

Backtracking

- Incrementally builds candidates to the solutions
- Abandons a candidate (backtrack) when does not unify in a subgoal
- It then reconsiders previous subgoal and tries to find another solution
- Multiples solutions results on different instantiations of a variable

Backtracking

```
male(X), parent(X, shelley).
```

This goal asks whether there is an instantiation of `X` such that `X` is a `male` and `X` is a `parent` of `shelley`. As its first step, Prolog finds the first fact in the database with `male` as its functor. It then instantiates `X` to the parameter of the found fact, say `mike`. Then, it attempts to prove that `parent(mike, shelley)` is true. If it fails, it backtracks to the first subgoal, `male(X)`, and attempts to resatisfy it with some alternative instantiation of `X`. The resolution process may have to find every `male` in the database before it finds the one that is a `parent` of `shelley`. It definitely must find all `males` to prove that the goal cannot be satisfied. Note that our example goal might be processed more efficiently if the order of the two subgoals were reversed. Then, only after resolution had found a `parent` of `shelley` would it try to match that person with the `male` subgoal. This is more efficient if `shelley` has fewer parents than there are `males` in the database, which seems like a reasonable assumption. Section 16.7.1 discusses a method of limiting the backtracking done by a Prolog system.

RefLang

- We add extensions to our language to support side effects
- These extensions focus on reading and writing memory locations
- We need two concepts and definitions
  - Heap: memory reserved for dynamic alloc
  - References: locations in the heap

Reference Expressions

- Represent malloc expressions
- (ref 1): stores the value 1 in a fresh location
- (free (ref 1)): deallocate the location for (ref 1)
- (deref (ref 1)): dereference a previously allocated memory location
  - $ (define loc (ref 3))
  - $ (deref loc)
  - $ 3

Reference Expressions

- (set! loc v): mutates the value of location *loc*, assigning value *v*
  - *$ (define loc (ref 5))*
  - *$ (deref loc)*
  - *$ 5*
  - *$ (set! loc 10)*
  - *$ (deref loc)*
  - *$ 10*

Examples RefLang

- $ (define loc (ref 5))
- $ (deref loc)
- $ 5
- $ (free loc)
- $ (deref loc)
- $ Error: null

Examples RefLang

- $ (define l1 (ref 10))
- $ (define l2 (ref 20))
- $ l2
- $??
- $ (free l2)
- $ (l2)
- $??
- $ (set! L2 20)
- $ 20
- $ (deref l2)
- $??
- $ (define l2 (ref 30)
- $ l2
- $??

Examples RefLang

- $ (define l1 (ref 10))
- $ (define l2 (ref 20))
- $ l2
- $ 1
- $ (free l2)
- $ (l2)
- $ Error: null
- $ (set! L2 20)
- $ 20
- $ (deref l2)
- $ 20
- $ (define l2 (ref 30)
- $ l2
- $ 30