CMPUT 325 Logic and constraint programming

Lecture Intro to Prolog

- logic programming is a theorem proving process
- length of a list:
  - ```
    len([], 0).
    len([First|Rest], N) :- len(Rest, NRest), N is NRest + 1.
    ```
  - predicate `len` - length of a list
- prolog program cosists of clauses
  - a clause can be a fact (unconditional) or a rule (conditional)
- lists
  - enclosed in `[ ... ]`
  - construct with a vertical bar `|`, or manually listing items, or a mix of both
    - `[First|Rest]`
    - `[a, b, c, d]`
  - can be nested
- variables and constants
  - case-sensitive
  - variables are in title-case, otherwise constant
  - free variables can match to anything upon a query
- why logic programming?
  - declarative style: style of building the structure and elements of a program
  - search process
  - control is more indirect
  - workflow: specify properties a solution should have, then let Prolog search for it
- syntax
  - rules about what are well-formed formulas in logic
- semantics
  - meaning of the program
  - description of all logical consequences of a formula
- inference rules
  - can be used to derive new formulas from the given set of formulas
- atoms in predicate calculus
  - `p(t1, ..., tn)`
    - `p` is a predicate symbol
    - `ti` are terms (like sexpr in Lisp)
      - can only be used are arguments in predicates
      - functions, constants and variables are terms
      - if `s1, ..., sk` are terms and `f` is a k-ary function symbol, then `f(s1, ..., sk)` is a term
- functions and predicates
    ```

- function symbols and predicates in Prolog start with a lowercase letter
- Binding variables
  - variables can be bound to values
  - use `=`, eg. `X = 2.`
  - there are other ways to bind variables using unification, coming up soon
  - two variables can be made equal without giving them value, `X = Y.`
    - Prolog will take that as a constraint
- rules
  - `:-` can be read as "if"
  - `parent(X, Y) :- father(X, Y)`
    - head = `parent(X, Y)`
    - body = `father(X, Y)`
  - head is true if its body is true or another predicate with the same head is true
  - in general 'A :- B1, B2, ..., Bn`
    - `,` in the body means "and"

Lecture Basic builtin operators and predicates in Prolog

- when predicates are used in a specific way, mark them in the specs:
  - **++**: ground: no variables
  - **+**: structure is clear
  - **-**: output parameter
  - **?**: param that can be used as either input or output
- anonymous variable: _ underscore
  - ```
    len([], 0).
    len([_|Rest], N) :- len(Rest, NRest), N is NRest + 1.
    ```
  - `First` is not needed in the `len` example above, replace it by an underscore to avoid `Singleton variables` warning
  - any variable that occurs only once in a rule should be anonymized like this
- Built-in operators - arithmetic and `is`
  - `Var is Expression`: evaluates expression, match with Var
  - operators include arithmetic operators (+, -, *, /), comparison operators (>, <, >= , =<)
    - `Var` can also be a constant
  - equality operators
    - `X = Y` tries to match X, Y equal by unification (pattern matching)
    - `X is Expr` evaluates `Expr`, then tries to make result equal to `X`
    - `T1 == T2`, are two terms currently identical? (no unification)
  - non-equality operators
    - `T1 \== T2`, are two terms not identical? (no unification)
    - `E1 =:= E2`, are `E1` and `E2` equal-valued arithmetic expressions
    - `E1 =\= E2`, are `E1` and `E2` different-valued arithmetic expressions?
  - `is`
    - evaluates and assigns
    - order of evaluation is leftmost first, when `X+5` is evaluated `X` is free
    ```

- Meta-logical predicates
  - `var(X)`: test whether `X` is instatiated
  - `nonvar(X)`: opposite of `var(.)`
  - `atom(X)`: check if `X` is instatiated to an atom
  - `integer(X)`
  - `number(X)`
  - `atomic(X)`: true if `X` is either an atom or number

Data structures in Prolog

- list predicates
  - ```
    append([], L, L).
    append([A|L1], L2, [A|L3]) :- append(L1, L2, L3).
    ```
    - ```
      append([a1, a2], [b1], A).
      A = [a1, a2, b1].
      ```
  - ```
    member(X, [X|_]).
    member(X, [_|L]) :- member(X, L).
    ```
  - ```
    not_member(_, []).
    not_member(X, [Y, L]) :- X \== Y, not_member(X, L).
    ```
  - Reverse a list
    - ```
      reverse([], []).
      reverse([A|Rest], Rev) :- reverse(Rest, RevRest), append(RevRest,
      [A], Rev).
      ```
- prolog clauses are NOT like if-then-else statements, they always use all matching clauses
  - set up exact conditions for when a clause should be used
  - Example:

  ```
  p(Input, result1) :- test(Input).
  p(Input, result2) :- opposite-of-test(Input).
  ```

  ```
  ```
  ```

```
p(Input, result1) :- test1.
p(Input, result2) :- opposite-of-test1, test2.
p(Input, result3) :- opposite-of-test1, opposite-of-test2, opposite-of-test3.
```

Prolog examples

-

```
sqsum([], 0).
sqsum([L|R], N) :- sqsum(R, NRest), N is NRest + L * L.
```

- `lastN(+L, +N, ?R)`
  - `L` is a given list
  - `N` is a given integer
  - `R` should be a list which contains the last `N` items in L
  - assume `N` is at least 0
  - Idea 1:
    - compute the length `LLength` of list `L`
    - compute the difference `D = LLength - N`
    - remove the first `D` items from `L` to get `R`

    ```
    lastN(L, N, R) :-
      length(L, LLength),
      D is LLength - N,
      removeFirstN(L, D, R).

    removeFirstN(L, 0, L).
    removeFirst([_|L], N, R) :-
      N > 0,
      N1 is N - 1,
      removeFirstN(L, N1, R).
    ```

  - Idea 2:

```
lastN(L, N, R) :-
  length(L, LLength),
  LLength > N,
```

```
    L = [_|Rest],
    lastN(Rest, N, R).
```

Finding all solutions

- we can use backtracking to find all solutions and store them in a list
- syntax: `findall(Variable, Goal, Solutions)`
- example: `findall(X, member(X, [a,a,b]), Result).`
  - output: `Result = [a, a, b].`
- collects all solutions for `X` in a list `Result`

Lecture Unification

- two way matching process to make two sides of an equation equal
- finds variable substitutions that make sides identical
- in Prolog, all variables are **local to the current clause**
- it is important
  - whether something **still is** a variable, or has been bound
  - whether two variables are the same, or not
- internally, prolog uses variables such `_number`

Substitution

- replace a variable by some other term
- a mapping *w = X1/t1, ..., Xn/tn*
  - X1, ..., Xn are distinct variables
  - t1, ..., tn are terms (objects composed of function symbols, variables and constants)
- Substitution `w = {X/b, Y/f(Z)}`
- Term `t = f(X, g(Y))`
- Applying substitution results in `w(t) = f(b, g(f(Z)))`
- each variable maps to exactly one term

Unifier

- definition: **unifier** of two terms `C1` and `C2`:
  - a substitution `w` such that `w(C1) = w(C2)`
  - a unifier of two terms makes them identical after the substitution
- two objects are unifiable if there exists a unifier for them
- example 1
  - ```
    t1 = f(a, X)
    t2 = f(Y, b)
    w = X/b, Y/a
    ```
  - `w(t1) = f(a, b)`
  - `w(t2) = f(a, b)`
    ```

    - therefore, `w(t1) = w(t2)`
    - `w` is a unifier of `t1` and `t2`
- example 2
    - ```
      t1 = p(f(X, X), Y)
      t2 = p(f(a, Z), b)
      w = {X/a, Z/a, Y/b}
      ```

  ```
  * ```
  t1 = p(X, X)
  t2 = p(a, b)
  no unifier exists ==> t1 and t2 are not  unifiable
  ```

Most General Unifier

- ```
  t1 = p(f(X, Y), Z)  t2 = p(Z, f(a, Y))
  w = {Z/f(a, Y), X/a} is a unifier
  w' = {Z/f(a, b), X/a, Y/b} is also a unifier
  ```
    - `w` is a more general unifier
    - `w'` can be obtained from `w` by adding more substitutions
        - converse does not hold however
- Theorem: For two unifiable terms, there always exists a **most general unifier**. It is **unique** upto renaming of variables.

Unification algorithm

- Given two terms `t1` and `t2`, generate MGU or prove they are not unifiable efficiently and exactly.
- Main idea:
    - match outside-most predicate-or-function and their arity
    - match each arg recursively
    - this will give us a system of equations at each step
        - the equations get simpler at each step as we "take apart" complex terms
    - keep track of substitutions that are needed
    - stop with failure if there is a contradiction
    - stop with success if all equations become identities, return the latest list of substitutions
- Example:
    - ```
      t1 = p(f(g(X,a)), X)
      t2 = p(f(Y), b)
      ```
    - system of equations `S0`
      ```

        - `S0 = {p(f(g(X, a)), X) == p(f(Y), b)}`
        - `w0 = {}`
    - both have name `p` and two args
    - match the args on each side
        - `S1 = {f(g(X, a)) == f(Y), X == b}`
        - `w1 = {}`
    - easier to solve the simplest equation first
        - `S2 = {f(g(b, a)) == f(Y), b == b}`
        - `w2 = {X/b}`
    - continue...
        - `S3 = {g(b, a) == Y, b == b}`
        - `w3 = {Y/g(b, a), X/b}`
    - finally we have an MGU
        - `S4 = {g(b, a) == g(b, a), b == b}`
        - `w4 = {Y/g(b, a), X/b}`
- this algorithm always generates a unique MGU, upto variable renaming
    - variable renaming means `X == Y` implies either `Y == Y` or `X == X`

Occurs check

- checking cases such as `Y == f(g(Y))` during unification is called the **occurs check**
- expensive operation
- many systems skip it to avoid overhead, but this makes the system not trustworthy

Inference engine of prolog - resolution and tree search

- **Horn clause**: a single atom (predicate) at the head
- Prolog uses only Horn clauses
- Setting
    - variables: identifier starting with an upper case letter
    - constants: identifier starting with a lower case letter
    - variables in different clauses are not related
    - lifetime of a variable is a single clause
    - variables in facts and rules are universally quantified
    - variables in queries are existentially quantified
- Notation for logic
    - exists
    - forall
    - entails (S follows logically from P)
    - negation
- what is resolution?
    - resolution is propositional logic without any variables

  - take two clauses c1 and c2 that are disjunctions of literals
  - c1 contains a variable v and c2 contains the literal ~v
  - disjunction of everything in c1 and c2, except v and ~v dropped
  - example: `c1 = ~a v b, c2 = a v c`
    - resolution of c1 and c2: `b v c`
- why is resolution valid?
  - prove by contradiction by assuming `b v c` is false
  - then `a ^ ~a` must be true, contradiction!
- Derived goal, resolution for Prolog
  - Goal *G*: `?- C1,...,Ck.`
  - A program clause: `A :- B1,...,Bn.`
  - Unification: `w = unify(A,C1)`
  - Derived goal `?- w(B1,...,Bn,C2,...,Ck).`
  - Special case: program clause is fact `A.`
  - Derived goal `?- w(C2,...,Ck).`
  - Another special case: Single goal, resolved by fact: derived goal is empty, done.
  - Meaning in terms of resolution:
    proof by contradiction is complete
  - Assuming $\sim G$ led to contradiction

Prolog's tree search

- Given a goal G: `?- C1, ..., Ck.`
- there can be a number of unifiable clauses whose head is unifiable with `C1`
- also, any subgoal can have several derived goals
- tree search:
  - root = original query
  - child of a node: a derived goal by resolution
  - each child node represents a resolution with a different clause in the program
  - **derivation** is a sequence of resolution steps
  - tree contains all possible derivations for a given program and a given goal
  - leaf node in the tree:
    - empty goal: success
    - failed branch: first subgoal is not unifiable with the head of any clause in the program
  - search tree can be infinite, then a path never reaches a leaf node
- `X/t` means variable `X` is bound to `t` by unification
- derivation sequence from original goal `p(Y)` to empty goal `[]`:

    - this represents a successful proof, called **refutation** (of the negation of the goal)
- DFS better than BFS in space complexity