CS 131 Programming Languages

November 3, 2009

Prolog and Homework 4

Here's an example kenken call:

```
kenken  (4,
        [+(3,[A1,B1]), *(12,[B2,B3,B4]), -(5, C1,C2)],
        [[A1,A2,A3,A4], [B1,B2,B3,B4], [C1,C2,C3,C4], [D1,D2,D3,D4]]).
```

Part of the above statement/structure can be parsed in Prolog if it's in a form like this:

```
B2*B3*B4 #=# 12        % constraint 2
```

There is a built-in predicate, `=..`, that converts constraints in this way:

```
*(12, [B3,B4,B5]) =.. [*, 12, [B3,B4,B5]]
```

Finite domain documentation describes how you can look and see the predicates that kenken call gives you and translate them into something that Prolog can understand. You can walk through the list of constraints, converting them on the go, and then ask prolog to prove the resulting output structure.

The command interpreter will pass to the kenken program a list of constraints that is *a list of lists*, and kenken program must walk through said list recursively, proving each one on the go.

Look at the finite domain solver and look at what predicates it gives you so you can get a handle on how to handle the task of solving the kenken puzzle. Every predicate which name starts with `fd` is a finite domain predicate. There's also a predicate that says "this variable has a range of 1 through 5." There's another predicate that says "these values must all be different."

Set up a two-dimensional array in which constraints are placed on each row and on each column. Since the caller already specified the variables down here and up here, that work has been done by the Prolog parser. Just set up the constraints for the array, and make sure the arithmetic constraints hold, and you're done.

Extra credit question: find out which way is it faster to apply predicates: uniqueness first or arithmetic first?

Is there an easy way in Prolog to extract columns from the given list of rows? *Transpose* predicate in GNU Prolog can help. You can write one or use the one built-in.

Debugging in Prolog

Often times it's handy to be able to print all solutions to a goal. Let's say you have something you're trying to test:

```
?- g(A,B,C).
A=12,B=f(12), C=B;…                          % more solutions avail, but have to continue to input a ";"
?- g(A,B,C), write(ans,(A,B,C)), nl, fail.   % write out each answer, then write a newline, then fail and
                                             % backtrack to g(A,B,C) until it fails (back and forth…)
```

What is a definition of *true*? One possibility is `true :- 1=1`. But a simpler way to define it as "`true.`"

There's a built-in Prolog predicate "equals," an operator of arity 2: `f(3,X) = f(Y,12)`. A longer notation for the same call is `=(f(3,X), f(Y,12))`. How can we define it? Equals can be thought of a gigantic pattern match operators where both sides are patterns. Prolog doesn't distinguish between patterns and data – everything that's data is also a pattern. We find the simplest substitution that makes the statement true. Here's a definition of *equals*:

```
=(X,X).                                      % can also write X=X.
```

A process of *unification* occurs when equal operator is used. One gives Prolog two terms and it comes up with consistent substitution values for variables that makes the terms identical. The substitution is a set of (name, value) pairs or a failure. Trees are constructed for each of the argument and Prolog tries to make the trees identical (hence the *equals* name).

Unification works both ways:

1. Data in the goal can become bound to terms.
    a. For example, `append([3], [7], Z)` results in `Z` being bound to `[3], [7]`.
    b. Calling to `append([3], [7], [X | Y])` will bind `3` to `X` and `7` to `Y`.
2. Variables in terms in a fact or rule (clause) can become bound to terms defined by the data. This is more like what ML does in pattern matching.

```
a.  p(X, f(X)).
        i.  p(g(37), Z).   % X is bound to g(37)
```

Here's an informal notation for an API:

```
foobar(+X, +Y, -Z).    % X and Y are input arguments that will be instantiated, Z is an output argument
% for input arguments, the intent is to bind variables in facts and rules and values in the arguments
% for output arguments, the intent is to bind the variables in output arguments to values in facts and rules
```

Here's another possible problem with unification. Let's have a predicate:

```
p(X,X).         % this is just an equal operator by another name
?- p(Y, f(Y))   % is Y = f(Y)?
        yes. Y = f(f(f(f(f…
```

Let's define our own arithmetic rules in Prolog. We'll use *Peano arithmetic*.

```
add(zero, X, X).                                    % zero = 0
add(s(X), Y, s(XplusY)) :- add(X, Y, XplusY).       % s(X) = x + 1
lt(zero,s(_)).                                      % zero < anything
lt(X, s(X)).                                        % x < x + 1
lt(X, s(Y)) :- lt(X, Y).                            % x < y + 1 if x < y

?- lt(I, I).                                        % is there a number less than itself?
I = s(s(s(s(s(…                                     % Prolog tells that infinity is less than itself.
```

`unify_with_ocurs_check(X, Y)` acts like *equals* but checks for loops and fails if a loop would be created. However, it's slower so it's not built-in to Prolog's ordinary *equals*. The cost of *equals* is the minimum of the size of two trees that it tries to unify: `O(min(M, N))`. The cost of `unify_with_occurs_check` is the maximum of the size of the two trees: `O(max(M, N))`.

An ordinary Prolog variable, as computation goes on, gets bound to a structure that places constraints to the future values of the variable. Therefore, when we bind A to Y where Y is g(X), we bind A to a structure that limits what A can be. This is a *structural constraint*.

With a constraint solver, we restrict A so it becomes a constrained variable of certain type (such as *integer*) and value (such as 1...9). This is a *value constraint*.

Here's a meta predicate that checks whether its last member of the variables list is true.

```
last_is_true([X]) :- X.
last_is_true([_ | L]) :- last_is_true(L).
```

Efficiency Problem

Suppose you have a big hairy goal big_hairy_goal(X, L), member(X, L), big_hairier_goal(X, L). There's an efficiency problem with this code. Suppose when we complete big_hairy_goal, X = 19 and L = [3, 19, 12, 19, 5, 6, 19, 19]. What will happen? Here's the source code to *member*:

```
member(X,  [X | _]).
member(X, [_ | L]) :- member(X, L).
```

After *big_hairier_goal* initially fails, the code will backtrack to *member* and member will call *big_hairier_goal* three more times when it matches X to L. This is inefficient because if *big_hairier_goal* fails, we'd like to backtrack into *big_hairy_goal* to find another solution instead of having *member* provide multiple matches and calling *big_hairier_goal* again.

A workaround is *cut* (!) which always succeeds when called but when backtracked into, the calling predicate immediately fails. Here's a new definition of member:

```
memberchk(X, [X | _]) :- !.
memberchk(X, [_ | L]) :- memberchk(X, L).
```

A Prolog computation is a depth first left to right search of a large tree. When we back out of the *cut*, it selectively prunes away parts of the search tree which makes the program go faster. We're not wasting time in places where we know there's no solution.

The *cut* operator can be dangerous when used in a wrong way. Here's a standard predicate built into GNU Prolog that is a rough approximation to *not*.

```
\+(P) :- P, !, fail.
\+ _.

?- \+ member(3, [2,5,7]).
yes
?- \+ member(3, [3,1,5]).
no
?- \+ member(X, [3,5]).        % is there X such that X is not a member of [3,5]
no                             % the answer should be "yes" since there are a lot of values other than 3 and 5
?- X=10, \+ member(X, [3,5]).
yes X=10
```

A good programming rule is to apply *not* to ground terms only (terms that have no logical variables). The logical variables lead to contradictory results like the above.

More generally, the not provable predicate works only if the closed-world assumption holds (I know everything that's true and I know everything that's false and there's nothing else).