The Seventh Chamber: Disjunction

Randall A. Helzerman

September 23, 2014

Contents

Chapter 1

Infrastructure for Seventh Chamber

```
*/

:- op(870,xfy,[=>]).
:- op(880,xfy,[<=>]).
:-op(900,fx,not).

:-discontiguous(np/6).
:-discontiguous(defunkify/3).
:-discontiguous(case/2).
:-discontiguous(case/2).
:-discontiguous(clause/1).
:-discontiguous(funky/1).
:-discontiguous(kind/3).
:-discontiguous(word/1).
:-discontiguous(mood/3).
:-discontiguous(process_logical_form/2).
:-discontiguous(sense/3).
:-discontiguous(sense/4).
:-discontiguous(voice/3).
:-discontiguous(sense/5).
:-discontiguous(sense/6).
:-discontiguous(subcat/3).
:-discontiguous(subcat/4).
:-discontiguous(grammatical_number/3).
:-discontiguous(grammatical_person/3).
:-discontiguous(gender/2).
:-discontiguous(category/2).
:-discontiguous(category/3).
:-discontiguous(category/4).
:-discontiguous(behavior/3).
:-discontiguous(verb_base/2).
```

```
:-discontiguous(verb_form/5).
:-discontiguous(prove/3).
:-discontiguous(quantifier/3).
:-discontiguous(root_word/2).
:-discontiguous(s/3).
:-discontiguous(conjugation/2).
:-discontiguous(thing_range/3).
:-discontiguous(thing_super/2).

:-dynamic(axiom/1).
:-dynamic(dial/3).

/*
```

1.0.1 Not predicate

```
*/

not F --> F, !, { fail }.
not _ --> [].

gensym(Root, Atom) :-
        name(Root,Name1),
        get_num(Root,Num),
        name(Num,Name2),
        append(Name1,Name2,Name),
        name(Atom,Name).

get_num(Root,Num1) :-
        retract(current_num(Root,Num)),!,
        Num1 is Num + 1,
        asserta(current_num(Root,Num1)).

get_num(Root,1) :- asserta(current_num(Root,1)).

skolemise(X) :- gensym(sk,X).

/*
```

1.1 Getting a line of input from stdin

```
*/

get_sentence(S) :-
        snarf_line(L1),
        chunks(_Dummy,S,L1,_L2).

snarf_line(L) :-
        %  Get a character.
        get0(C),

        % process it.
        ( (C = 10) ->
            L = []
        ;
            snarf_line(Cs),
            L = [C|Cs]
        ).

/*
```

1.2 Chunks

Now, we break up the input list into ”chunks”:

```
*/

chunk([A|L]) -->
        optional_white_space,
        apostrophe(A),
        chunk(L).

chunk(W) -->
        optional_white_space,
        word(W).

chunk(W) -->
        optional_white_space,
        numeral(W).

chunk([W]) -->
        optional_white_space,
        non_chunking_punctuation(W).

/*
```

1.2.1 Whitespace

Chunks are seperated by whitespace, so we have to specify what counts as whitespace:

```
*/

white_space([H|T]) -->
        white_space_char(H),
        (
          white_space(T)
        ;
```

```
          end_white_space(T)
        ).

white_space_char(H) -->
        (
          space_char(H)
        ;
          tab_char(H)
        ).

space_char(32) --> [32].
tab_char(9)    --> [9].

end_white_space([]) --> not white_space_char(_H).

/*
```

We use this trick for optional white space:

```
*/

optional_white_space --> white_space(_S),!.
optional_white_space -->[].

/*
```

1.2.2 Recognizing Letters

```
*/

letter(H) -->
        (
          lower_case_letter(H)
        ;
          upper_case_letter(H)
        ;
          underscore(H)
        ).


lower_case_letter(H) -->
        [H],
        { H >= 97  },
        { H =< 122 }.

upper_case_letter(H) -->
        [H],
        { H >= 65 },
        { H =< 90 }.
```

```
underscore(95) --> [95].

/*
```

1.2.3 Words

The next step is to group segments of letters into words.

```
*/

end_word([]) --> not letter(_H).
word([H|T]) -->
        letter(H),
        ( word(T)
        ; end_word(T)
        ).


/*
```

1.2.4 Numerals

In the same way, we can group digits and numbers:

```
*/

digit(H) -->
        [H],
        { H >= 48 },
        { H =< 57 }.

end_numeral([]) --> not digit(_H).
numeral([H|T]) -->
        digit(H),
        (
          numeral(T)
        ;
          end_numeral(T)
        ).

/*
```

1.2.5 Punctuation

Words can also be seperated by punctuation:

```
*/

punctuation(C) -->
        apostrophe(C)
```

```
    ;
        non_chunking_punctuation(C).


non_chunking_punctuation(C) -->
        comma(C)
    ;
        semicolon(C)
    ;
        period(C)
    ;
        exclamation_mark(C)
    ;
        question_mark(C)
    ;
        hyphen(C).

comma(44) --> [44].

semicolon(59) --> [59].

apostrophe(39) --> [39].

period(46) --> [46].

exclamation_mark(33) --> [33].

question_mark(63) --> [63].

hyphen(45) --> [45].


/*
```

1.2.6 Defunkification

So-called "funky" words like "gonna" replaced by "gon na"

```
*/

funky(gunna).
funky(woulda).
funky(coulda).
funky(shoulda).
funky(something).
funky(somethings).

defunkify(gunna,[gun,na|X],X).
defunkify(woulda,[would,a|X],X).
defunkify(coulda,[could,a|X],X).
defunkify(shoulda,[should,a|X],X).
defunkify(something,[some,thing|X],X).
defunkify(somethings,[some,things|X],X).

/*
```

The reason we break things up like this is explained on the Penn treebank web page, "This tokenization allows us to analyze each component separately, so (for example) "I" can be in the subject Noun Phrase while "'m" is the head of the main verb phrase."

The following arn't funky in the sense of the Penn treebank, but since we already have the machinerty there, it is convenient to treat them as funky.

```
*/

funky(table).
defunkify(table,[w_table|X],X).

/*
```

1.2.7 Putting the chunks together

The following is really interesting...because a single chunk can induce more than one word in the resulting tokenized stream, we need to use append. To make that fast, we need to use difference lists. This makes the code much harder to understand...

```
*/


chunks(_H,[]) --> not chunk(_W).
chunks(H,T1) -->
        optional_white_space,
        chunk(H1),
        { name(H2,H1) },
        { ( funky(H2) ->
              defunkify(H2,T1,T2)
          ;
              T1 = [H2|T2]
          )
        },

        chunks(H,T2).
/*

*/

% colors are mutuallly exlcusive
clause((false :- green(X), red(X))).
clause((false :- green(X), blue(X))).
clause((false :- red(X), blue(X))).

% so are blockhood and tablehood
clause((false :- block(X), table(X))).
/*
```

Sharp-eyed readers can already see a problem with this technique, don't worry, we'll present a comprehensive solution later. The upside is that it is now quite easy to write our desired predicate:

```
*/
coherent(L) :- not prove(false,L).
/*
```

in addition, the last clause has been modified so that it simply skips atoms of the form "axiom(true)."

1.3 Dialog

The proceedure mmm_update needs to have a list of formulas we already know about, as well as a list of objects we already know about. Since we store these in the form of asserted axiom/2 predicates, we'll need some routines to assemble the list of formula and extract the known objects.

```
*/

extract_objects_from_formulas(Fs, Os2) :-
extract_objects_from_formulas_aux(Fs,Os1),
sort(Os1,Os2).

extract_objects_from_formulas_aux([],[]).
extract_objects_from_formulas_aux([F|Fs],Os3) :-
extract_objects_from_formula(F,Os1),
extract_objects_from_formulas_aux(Fs, Os2),
append(Os1,Os2,Os3).

extract_objects_from_formula(F,Os) :- F=..[_|Os].
/*
```

now we can write the dialog function:

```
*/
dialog :-
        % remove previous resuls
        retractall(axiom(_)),

        read_eval_loop.
```

```
read_eval_loop :-
        get_sentence(S),
        ( S=[bye,'.'] ->
            true
        ;
            process_sentence(S),
            read_eval_loop
        ).

process_sentence(S) :-
        % parse the sentence
        ( sentence(Mood,L,S,[]) ->
             % see how to process it
    process_logical_form(Mood,L)
        ;
    write('couldn\'t understand this:'),
    writeln(S)
).


% if this is a query, run the lloyd-topor transform and query
process_logical_form(yn_question,L) :-
lt_transform(L,Q),

writeln(prove(Q,[])),

(  prove(Q,[]) ->
    writeln('yes')
;
            writeln('no')
).
/*
```

1.4 Lambda Calculus

1.5 Developing Alpha Conversion and Beta Conversion using Skeletons and Techniques

This parser is the skeleton we use:

1.5.1 Syntax of Lambda Expressions

```
*/

lambda_expression(E) :- var(E),!.

lambda_expression(F*A) :- !,
        lambda_expression_application(F,A).

lambda_expression(X^F) :-!,
        var(X),
        lambda_abstraction(X,F).

lambda_expression(E) :-
        E =.. [_F|Ts],
        map_lambda_expression(Ts).

lambda_expression_application(F,A) :-
        lambda_expression(F),
        lambda_expression(A).

lambda_abstraction(_X,F) :-
        lambda_expression(F).

map_lambda_expression([]).
map_lambda_expression([T|Ts]) :-
        lambda_expression(T),
        map_lambda_expression(Ts).

/*
```

1.5.2 Bound Variables

To the above skeleton, we apply a techique. The technique is to recognize when a binding expression has been encountered, and save the variable which has been bound in a bag.

So, the first thing we do is think about the bag we want to store the bound variables in. We'll put them in a list, but if we want to test if a variable is a member off that list, we can't just use member/2, because member/2 will unify. So we write our own member-esque function which uses == instead of unification:

```
*/

variable_in_bag(X1,[X2|_]) :- X1 == X2.
variable_in_bag(X,[_|T]) :- variable_in_bag(X,T).

/*
```

Now for the visitor itself. We rename "lambdaExpression" to "visitBound-Variables" and add an extra argument which holds, at any point in the lambda expression, a list of the variables which are bound.

```
*/

visit_bound_variables(E, Bs) :- var(E),!,
        write(E), write(' is '),
        ( variable_in_bag(E,Bs) ->
            writeln(' bound')
        ;
            writeln(' free')
        ).

visit_bound_variables(F*A, Bs) :- !,
        visit_bound_variables_application(F,A, Bs).

visit_bound_variables(X^F, Bs) :-!,
        var(X),
        visit_bound_variable_abstraction(X,F,Bs).

visit_bound_variables(E, Bs) :-
        E =.. [_F|Ts],
        map_visit_bound_variables(Ts, Bs).

visit_bound_variables_application(F,A, Bs) :-!,
        visit_bound_variables(F, Bs),
        visit_bound_variables(A, Bs).

% notice that when a variable binder is encountered, it packs
% the variable on top of stack of bound variables
visit_bound_variable_abstraction(X,F,Bs) :-
        visit_bound_variables(F, [X|Bs]).

map_visit_bound_variables([], _Bs).
map_visit_bound_variables([T|Ts], Bs) :-
        visit_bound_variables(T, Bs),
        map_visit_bound_variables(Ts, Bs).

/*
```

1.5.3 Alpha Conversion

The bound variable visitor can in turn be used as a skeleton. Alpha conversion is the process of changing the names of the bound variables.

The first step is to make a data structure which we can use to associate one variable with another. This is also derived using skeletons and techniques; the skeleton being the variable_in_bag/2 procedure above. Instead of just containing variables, the bag will contain elements of the form sub(X,Y), which indicates that Y should be substituted for X.

```
*/

substitute_bound_variable(sub(X1,Y),[sub(X2,Y)|_]) :- X1 == X2.
substitute_bound_variable(X,[_|T]) :- substitute_bound_variable(X,T).
```

```
/*
```

We then add another parameter to the visit_bound_variables predicate–an output parameter. The alpha conversion process then works as a transducer, walking over the input lambda expression, substituting a fresh variable for every bound variable, and passing everything else along unchnaged to the output parameter:

```
*/

alpha_convert(F1, F2) :- alpha_convert(F1,[], F2).

alpha_convert(X1, Bs, X2) :- var(X1),!,
        ( substitute_bound_variable(sub(X1,X2),Bs) ->
            true
        ;
            X2=X1
        ).

alpha_convert(F1*A1, Bs, F2*A2) :- !,
        alpha_convert_application(F1,A1, Bs, F2,A2).

alpha_convert(X1^F1, Bs, X2^F2) :-!,
        var(X1),
        alpha_convert_abstraction(X1,F1, Bs, X2,F2).

alpha_convert(F1, Bs, F2) :-
        F1 =.. [Op|Ts1],
        map_alpha_convert(Ts1, Bs, Ts2),
        F2 =.. [Op|Ts2].

alpha_convert_application(F1,A1, Bs, F2,A2) :-
        alpha_convert(F1,Bs,F2),
        alpha_convert(A1,Bs,A2).

alpha_convert_abstraction(X1,F1, Bs, X2,F2) :-
        alpha_convert(F1, [sub(X1,X2)|Bs], F2).

map_alpha_convert([],_,[]).
map_alpha_convert([T1|Ts1], Bs, [T2|Ts2]) :-
        alpha_convert(T1,Bs,T2),
        map_alpha_convert(Ts1,Bs,Ts2).

/*
```

1.5.4 Free Variables

Free variables are variables appearing in a lambda expression which arn't bound. If we want to collect these, the easiest way is to also collect the bound variables, and whenever we run across a variable, check to see if it is a bound variable. If its not, then its a free variable, so we collect it.

So it would be plausible to use the boundVariable routine as a skeleton. However, there's another consideration. Consider the formula:

```
foo(X,X^bar(X))
```

Should our predicate indicate that X is free or bound? Its free in some contexts, and its bound in others.

The easiest (and safest) way out of this is to punt: Since the names of bound variables are arbitrary, why not just use alpha conversion to rename them? That way, we're guarunteed that none of the variables reported as free are bound in any context.

We could perform alpha conversion before collecting the free variables, but then we'd have to make two passes over the input lambda term. This extra pass can be avoided simply by choosing as a skeleton the alpha conversion predicate.

The technique is then to add a difference-list to the predicate to collect the free variables. The reason it is Perierran instead of fregian is that the returned list of free variables is the appendage of the lists from the subterms, so to get rid of the overhad of appending we use different lists.

```
*/

free_variables(F1, Us, F2) :- free_variables(F1, [], [],Us, F2).

free_variables(X1, Bs, U1,U2, X2) :- var(X1),!,
        ( substitute_bound_variable(sub(X1,X2),Bs) ->
            U1=U2
        ;
            X2=X1,
            U2=[X1|U1]
        ).

free_variables(F1*A1, Bs, U1,U2, F2*A2) :- !,
        free_variables_in_application(F1,A1, Bs, U1,U2, F2,A2).

free_variables(X1^F1, Bs, U1,U2, X2^F2) :-!,
        var(X1),
        free_variables_in_abstraction(X1,F1, Bs, U1,U2, X2,F2).

free_variables(F1, Bs, U1,U2, F2) :-
        F1 =.. [Op|Ts1],
        map_free_variables(Ts1, Bs, U1,U2, Ts2),
        F2 =.. [Op|Ts2].

free_variables_in_application(F1,A1, Bs, U1,U3, F2,A2) :-
        free_variables(F1, Bs, U1,U2, F2),
        free_variables(A1, Bs, U2,U3, A2).
```

```
free_variables_in_abstraction(X1,F1, Bs, U1,U2, X2,F2) :-
        free_variables(F1, [sub(X1,X2)|Bs], U1,U2, F2).

map_free_variables([], _, U,U, []).
map_free_variables([T1|Ts1], Bs, U1,U3, [T2|Ts2]) :-
        free_variables(T1, Bs, U1,U2, T2),
        map_free_variables(Ts1, Bs, U2,U3, Ts2).

/*
```

1.5.5 Alphabetic Variants

```
*/

alphabetic_variants(A1,B1) :-
        % grab the free variables in both the terms
        free_variables(A1,FA,A2),
        free_variables(B1,FB,B2),

        % see if the free variables are the same
        FA == FB,

        % instantiate all of the variables.
        % note: we cant stipulate that N1 is the same as
        % N2, because if A2 and B2 share free variables,
        % they will be instantiated during the first numervars
        % call and the second will indicate a different number of
        % variables...
        numbervars(A2,0,_N1),
        numbervars(B2,0,_N2),

        % see if the resulting terms are the same
        A2=B2.

/*
```

1.5.6 Beta Conversion

Beta conversion is the process of applying beta reduction over and over again until there's no more beta conversion which can be done.

```
*/

% driver
beta_convert(X,Y) :- var(X),!, X=Y.

beta_convert(E,R) :- beta_convert(E, [], R).
beta_convert(X,_S,X) :- var(X),!.

beta_convert(F*A, S, R) :- !,
        alpha_convert(F,CF),
        beta_convert_application(CF,A, S, R).

beta_convert(X^F, S, R) :-!,
        var(X),
        beta_convert_abstraction(X,F, S, R).

beta_convert(E, S, R2) :-
        E =.. [F|Ts],
        map_beta_convert(Ts, Rs),
        R1 =..[F|Rs],
        reapply_stack(S,R1,R2).

reapply_stack([],R,R).
reapply_stack([A|S],R1,R2) :-
        reapply_stack(S,(R1*A),R2).

beta_convert_application(X,A, _S, X*A1) :- var(X),!,
        beta_convert(A,A1).

beta_convert_application(F,A, S, R) :-
        beta_convert(F, [A|S], R).


beta_convert_abstraction(X,F, [], X^R) :-
        beta_convert(F,[],R).

beta_convert_abstraction(X,F, [X|S], R) :-
        beta_convert(F,S,R).


map_beta_convert([], []).
map_beta_convert([T|Ts], [R|Rs]) :-
        beta_convert(T, R),
        map_beta_convert(Ts, Rs).

/*
```

Chapter 2

The Seventh Chamber: Disjunction

2.1 Problems to Solve

We want to be able to have a conversation with the computer which looks like this:

```
The block is blue or red.
The table is red.
```

and then be able to answer questions like this:

```
Is the table red?
yes.

Is the block yellow?
no.

Is the block red?
maybe.
```

2.2 Grammar

Note, we'll be rewinding a bit here, because in this chapter we won't start from a language which includes conjunction. As will soon be obvious, disjunction introduces enough wrinkles of its own, and focusing exlucively on disjunction will allow us to more directly address those particular problems. The complexifications resulting from the interactions between conjunction and disjunction will be the task of the next chapter.

We won't be rewinding all the way, however. The machinery we've built up for implementing a compositional semantics with lambda calculus is still quite useful, so we'll keep that, we'll just simplify the production for noun phrases so that there's no conjunction. Basically, the grammar shown we'll be is quite a bit like the grammar **??**, with "and" changed to 'or' (I also has a few other changes, but they are minor).

```
*/
sentence(M,F) -->
s(M,L),
{ beta_convert(L,F) }.


% the block is green.
s(assertion,NPL*VPL) -->
np(Number,3,NPL),
vp(Number,[],[], VPL),
['.'].

% there is a block.
s(assertion,NPL*VPL) -->
advp([],[],AvpF),
verb(Number,[],[],W),
np(Number,2,NPL),
vp(Number,[gap(W,Number),gap(advp,AvpF)],[], VPL),
['.'].

% is the block green?
s(yn_question,NPL*VPL) -->
verb(Number,[],[],W),

np(Number,3,NPL),

vp(Number,[gap(W,Number)],[], VPL),

['?'].

% is there a block?
s(yn_question,NPL*VPL) -->
verb(Number,[],[],W),
advp([],[],AvpF),
np(Number,3,NPL),
vp(Number,[gap(W,Number),gap(advp,AvpF)],[], VPL),

['?'].


np(singular,C,D*N) -->
{ C>0 },
det(D), noun(N).

np(singular, C,D*(X^and(A*X,N*X))) -->
```

```
{ C>0 },
det(D), adjp(3,A),  noun(N).

np(singular, C,CL*N1*N2) -->
{ C>1 },
np(_Number1,1,N1),

conjunction(CL),

{ C1 is C-1 },
np(_Number2,C1,N2).


det(R^Q^exists(X,and(R*X,Q*X)))--> [a].
det(R^Q^exists(X,and(R*X,Q*X)))--> [the].


noun(X^block(X)) --> [block].
noun(X^w_table(X)) --> [w_table].


% is green
vp(Number,P1,P2,L) --> verb(Number,P1,P2,_), adjp(3,L).

% is there
vp(Number,P1,P3,L) --> verb(Number,P1,P2,_), advp(P2,P3,L).


verb(singular, [],[],is) --> [is].
verb(N,[gap(W,N)|P],P,W) --> [].


adjp(C, L) --> { C>0 }, adjective(L).

adjp(C, CL*L1*L2) -->
    { C>1 },
    adjp(1, L1),

    conjunction(CL),

    {C1 is C-1},
    adjp(C1, L2).


adjective(X^red(X))     -->  [red].
adjective(X^green(X))   -->  [green].
adjective(X^yellow(X))  -->  [yellow].


advp(P,P, F) --> adv(F).
advp([gap(advp,F)|P],P, F) --> [].


adv(_^true) --> [there].

% the whole point
conjunction(P^Q^X^or(P*X,Q*X)) --> [or].

/*
```

2.3 Representing disjunctive information

By convention, a prolog program describes one world, its least Herbrand model. But what about a theory which contains disjunctions? E.g.:

```
red(a) ; green(a)
```

if we restrict the above theory to just one model, we really are missing some important information.

So what we'll do is represent the alternative possibilities as alternate possible worlds. Each world will still be a least Herbrand model for some horn clause theory.

For example, upon reading in the sentence:

```
There is a red or a green block.
```

we would consider *two* worlds as being possible, and represent each world with a seperate Herbrand Model. The first model would be:

```
block(block1).  red(block1).
```

and the second would be

```
block(block1).  green(block1).
```

So how should we actually realize this in a prolog program? We could take a pointer from, say, situation calculus, and relativize each predicate to a particular world. For the example above, if we reify the two worlds (calling them, say, "w1" and "w2"), we could represent them as a single horn clause theory:

```
block(block1,w1).  green(block1,w1).
block(block1,w2).  green(block1,w2).
```

The problem with this method of representation is that it is rather prolix–e.g. notice that the blookhood of block1 has to be explicitly restated for each possible world.

Instead, lets factor out the facts common to all the worlds we consider possible, and explicitly represent the multiplicity *only* for facts which are disjunctive. We'll use the following metaphor. Consider the block which is either red or green. Consider a cosmic switch which has two positions. When the switch is in the first position, the block is red. When the switch is in the second position, the block is green. Our ignorance of whether the block is red or green therefore is our ignorance as to the position the switch is in.

In general disjunctions can have more than two possibilities, so instead of using binary switches, we'll use dials. A dial has a pointer which can be set any of several discrete positions, numbered 1 through n. Suppose that 'sk_dial' is the name of a dial, and we want to represent the fact that it can be in one of n positions. Using the predicate 'possible_dial_value/2', we can indicate it like this:

```
possible_dial_value(sk_dial,1).
possible_dial_value(sk_dial,2).
                 ⋮
possible_dial_value(sk_dial,n).
```

We can represent the fact that sk_dial's pointer is in position number 2, with the following prolog fact:

```
dial(sk_dial,2).
```

Consider our blocks world example. Lets make a dial for the block called 'block1_dial'. We'll represent the fact that if the dial is in position 1, the block is red and if the dial is in position 2, the block is green, with the following prolog rules:

```
block(block1).
possible_dial_value(block1_dial,1).
possible_dial_value(block1_dial,2).
green(block1)  :-  dial(block1_dial,1).
red(block1)    :-  dial(block1_dial,2).
```

Notice, the only fact we can derive from this prolog program (aside from the possible values for the dial) is that block1 is a block. We'll have to augment prolog's inferential capability in order to be able to conclude things like "the block *might* be green."

We'll do this with an abducing metainterpreter. The first few clauses of this meta-interpreter are standard vanilla (augmented with a few extra clauses for proving disjunctions).

```
*/
prove(A,R,R) :- axiom(A).
prove((A;_B),R1,R2) :- prove(A,R1,R2).
prove((_A;B),R1,R2) :- prove(B,R1,R2).
prove((A,B),R1,R3) :- prove(A,R1,R2),prove(B,R2,R3).
prove((A),R1,R2) :- clause((A:-B)), prove(B,R1,R2).
prove(not(A),R,R) :- \+ prove(A,R,_).
/*
```

So if we wrap the above horn clause theory with the arpopos 'clause' and 'axiom' predicates, we get:

```
*/
axiom(block(block1)).
possible_dial_value(block1_dial,1).
possible_dial_value(block1_dial,2).
clause((green(block1)  :-  dial(block1_dial,1))).
clause((red(block1)    :-  dial(block1_dial,2))).
/*
```

The trick is in this clause:

```
*/
prove(dial(D,V), R1,R2) :-
        ground(D),
        ( member(dial(D,V1),R1) ->
    V1=V,
            R2=R1
        ;
    possible_dial_value(D,V),
            R2=[dial(D,V)|R1]
        ).
/*
```

This clause does several things. First, it makes any fact of the form:

```
dial(D,V)
```

abductible. But, crucially, it also enforces that we never assume that a dial is in more than one position, i.e. we never assume two facts of the form:

```
dial(sk_dial,v1).
dial(sk_dial,v2).
```

so how do we use it? Well, we can now prove things like:

```
?- prove(block(X),[],R).
X = block1,
R = [] ;
```

which indicates that block1 is indeed a block. Moreover, since R is equal to the empty list, this means that in *all* possible worlds, block1 is a block. What happens if we try to prove that block1 is green?

```
?- prove(green(block1),[],R).

R = [dial(block1_dial, 1)] ;
```

In this case R is instantiated to a particular dial value, which indicates that block1 is green only in the possible world where block1_dial is pointing at 1.

```
*/

prove(A) :- axiom(A).

prove(A) :-
        prove(A,[],Rs),
        get_all_other_scenarios(Rs,Rss),
        prove_under_all_scenarios(Rss,A).

prove((A,B))  :- prove(A),prove(B).
prove((A))    :- clause((A:-B)), prove(B).
prove(not(A)) :- \+ prove(A,[],_).


get_all_other_scenarios(Rs,Rss) :-
        sort(Rs,Rs1),
        get_dials(Rs1,Ds),
        possible_scenarios_for_dials(Ds,Rss1),
        select(Rs1,Rss1,Rss).

prove_under_all_scenarios([],_A).
prove_under_all_scenarios([Rs|Rss],A) :-
        prove(A,Rs),
        prove_under_all_scenarios(Rss,A).


get_dials(Rs,Ds) :-
        get_dials_from_scenarios(Rs,Ds1),
        sort(Ds1,Ds).


get_dials_from_scenarios([],[]).
get_dials_from_scenarios([dial(D,_)|Rs],[D|Ds]) :-
        get_dials_from_scenarios(Rs,Ds).


possible_scenarios_for_dials(Ds,Rss) :-
        possible_assignments_for_dials(Ds,As),
        findall(Rs,possible_scenario_for_dials(As,Rs), Rss).

possible_scenario_for_dials([],[]).
possible_scenario_for_dials([D|Ds],[A|As]) :-
        select(A,D,_),
        possible_scenario_for_dials(Ds,As).


possible_assignments_for_dials([],[]).
possible_assignments_for_dials([D|Ds],[Rs|Rss]) :-
        possible_assignments_for_dial(D,Rs),
        possible_assignments_for_dials(Ds,Rss).
```

```
possible_assignments_for_dial(D,Rs) :-
ground(D),
findall(dial(D,V),possible_dial_value(D,V),Rs).

/*

*/
prove(A,_R) :- axiom(A).
prove((A;_B),R) :- prove(A,R).
prove((_A;B),R) :- prove(B,R).
prove((A,B),R) :- prove(A,R),prove(B,R).
prove((A),R) :- clause((A:-B)), prove(B,R).
prove(not(A),R) :- \+ prove(A,R). % this one has no abduction at all
prove(dial(D,V),R) :-
        ground(D),
        member(dial(D,V),R).
/*

*/

clause((wooden(X) :- red(X))).
clause((wooden(X) :- green(X))).

/*
```

What about the whole maybe thing? If we can't prove either p or not(p), then p is maybe true. This is the invokage of a generalized closed worlds assumption.

2.4 Compiling

Now that we have an inference engine which is capable of reasoning about this sort of disjunctive information, we need to address the problem of compiling and updating knowledge stored in the format that the inference engine can use.

```
*/

clause_form(F,C) :-
        cnf(F,F1),
        clausify(F1,C).

cnf(F1,F3) :-
        q_out(F1,F2),
        distribute(F2,F3).

q_out(not(F1), not(F2)) :-!, q_out(F1, F2).
q_out(and(A1,B1), and(A2,B2)) :-!, q_out(A1,A2), q_out(B1,B2).
q_out(or(A1,B1), or(A2,B2))  :-!, q_out(A1,A2), q_out(B1,B2).
q_out(exists(X,F1), F2) :-!,
skolemise(X),
q_out(F1,F2).

q_out(F,F).

/*
```

2.4.1 Distribute OR

```
*/

distribute(F1,F3) :-
        distribute(F1,F2,C),
        ( (C=true) ->
            distribute(F2,F3)
        ;
            F2=F3
        ).

distribute(or(and(A,B), F), and(or(A,F),or(B,F)), true) :-!.
distribute(or(F,and(A,B)),  and(or(F,A),or(F,B)), true) :-!.
distribute(or(A1,B1),       or(A2,B2), C) :-!,
    distribute(A1,A2,C1),
    distribute(B1,B2,C2),
    dor(C1,C2,C).

distribute(and(A1,B1),       and(A2,B2), C) :-!,
    distribute(A1,A2,C1),
    distribute(B1,B2,C2),
    dor(C1,C2,C).

distribute(F,F,false).

dor(false,false,false).
dor(false,true,true).
dor(true,false,true).
dor(true,true,true).


/*
```

2.4.2 Flatten to clauses

```
*/
clausify(F1,F3) :-
        clausify_and(F1,F2,[]),
        clausify_or(F2,F3).
```

```
clausify_and(and(A,B), L1,L3) :-!,
        clausify_and(A, L1,L2),
        clausify_and(B, L2,L3).
clausify_and(F, [F|T], T).

clausify_or([],[]).
clausify_or([H1|T1],[H2|T2]) :-
        clausify_or(H1,H2,[]),
        clausify_or(T1,T2).

clausify_or(or(A,B), L1,L3) :-!,
        clausify_or(A, L1,L2),
        clausify_or(B, L2,L3).
clausify_or(F, [F|T], T).
/*
```

2.4.3 Variate the Constants

```
*/

variate_constants(C1,C2, Vs) :-
variate_constants(C1,C2, [],B),
strip_vars(B,Vs).

variate_constants([],[], B,B).
variate_constants([C1|Cs1],[C2|Cs2], B1,B3) :-
variate_constants_in_clause(C1,C2, B1,B2),
variate_constants(Cs1,Cs2, B2,B3).

variate_constants_in_clause([],[], B,B).
variate_constants_in_clause([P1|Ps1],[P2|Ps2], B1,B3) :-
variate_constants_in_predicate(P1,P2, B1,B2),
variate_constants_in_clause(Ps1,Ps2, B2,B3).

variate_constants_in_predicate(P1,P2, B1,B2) :-
P1 =.. [F|F1],
variate_list(F1,F2, B1,B2),
P2 =.. [F|F2].

variate_list([],[], B,B).
variate_list([H1|T1],[H2|T2], B1,B3) :-
( var(H1) ->
    H2=H1,
    B2=B1
;
    ( member((H1,New),B1) ->
H2=New,
B2=B1
    ;
H2=New,
B2=[(H1,New)|B1]
    )
),
variate_list(T1,T2, B2,B3).

strip_vars([],[]).
strip_vars([(_,V) |T], [V|Vs]) :- strip_vars(T, Vs).

/*
```

2.4.4 The Maxim of Minimum Mutilation

```
*/

process_logical_form(assertion,L) :-
% process input sentence
c_transform(L,Fs1,Vs),
xormal_form(Fs1,Fs2),

% process previously-asserted database
assemble_formulas(Db1),
retractall(axiom(_)),
retractall(dial(_,_,_)),
        extract_objects_from_formulas(Db1,Os),

% calulate what the new db should look like
mmm_update(Vs,Os,Fs2, Db1,Db2),

% and assert it.
assert_all_as_axioms(Db2).

assemble_formulas(Cs,Xs) :-
assemble_clauses(Cs),
assemble_xors(Xs).

assemble_clauses(Cs) :- findall([F],axiom(F),Cs).

assemble_xors(Xs) :-
% first, find out what switches there be.
setof(N,A^V^dial(A,N,V),Ns),
map_get_xors_from_dials(Ns,Xs).

map_get_xors_from_dials([],[]).
map_get_xors_from_dials([H1|T1],[H2|T2]) :-
get_xors_from_dial(H1,H2),
map_get_xors_from_dials(T1,T2).

get_xors_from_dial(D,Xs) :-
possible_values_for_dial(D,Vs),
map_get_disxunct_for_dial_value(D,Vs,Xs).

map_get_disxunct_for_dial_value(_D,[],[]).
map_get_disxunct_for_dial_value(D,[V|Vs],[X|Xs]) :-
get_disxunct_for_dial_value(D,V,X),
map_get_disxunct_for_dial_value(D,Vs,Xs).

get_disxunct_for_dial_value(D,V,X) :- findall(A,dial(A,D,V),X).


possible_values_for_dial(D,Rs) :- setof(V,X^dial(X,D,V),Rs).


assert_all_as_axioms([]).
assert_all_as_axioms([F|Fs]) :-
assert(axiom(F)),
assert_all_as_axioms(Fs).



mmm_update(Vs, Os, Fs, Db1,Db3) :-
generate_domains(Vs,Os,Fs,Db1, Doms),
find_simultainously_consistent_assignment(Doms,Fs,Db1,Db2),
sort(Db2,Db3).

generate_domains([], _Os,_Fs,_Db1, []).
generate_domains([V|Vs], Os,Fs,Db1, [domain(V,Pos)|Doms]) :-
find_formulas_about_variable(Fs,V,Vfs),
append(Vfs,Db1,F1s),
find_possible_values(Os,V,F1s, Pos),
generate_domains(Vs, Os,F1s,Db1, Doms).

find_formulas_about_variable([],_V,[]).
find_formulas_about_variable([F|Fs],V,R) :-
F=..[_Functor|Args],
( variable_member(V,Args) ->
    R = [F|Vfs]
;
    R=Vfs
),
find_formulas_about_variable(Fs,V,Vfs).

variable_member(V,[VT|Vs]) :-
    (
        V==VT
    ;
        variable_member(V,Vs)
    ).
```

```
find_possible_values(Os,V,Fs,Pos) :-
    find_possible_preexisting_values(Os,V,Fs,Pos1),
    ( []=Pos1 ->
        skolemise(X),
        Pos=[X]
    ;
        Pos=Pos1
    ).


find_possible_preexisting_values([],_V,_Fs, []).
find_possible_preexisting_values([O|Os],V,Fs,Pos) :-
    ( \+ \+ (V=O, coherent(Fs)) ->
          Pos = [O|Pos1]
    ;
          Pos=Pos1
    ),
    find_possible_preexisting_values(Os,V,Fs,Pos1).


find_simultainously_consistent_assignment(Doms,Fs,Db1,Db2) :-
     append(Fs,Db1,Db2),
     assignment(Doms),
     coherent(Fs).

assignment([]).
assignment([domain(X,D)|Doms]) :-
    select(X,D,_D1),
    assignment(Doms).

/*
```

2.4.5 Lloyd-Topor Transform

```
*/

lt_transform(exists(_X,R), R1) :- !,
        lt_transform(R,R1).

lt_transform(and(true,Q), Q1) :- !,
        lt_transform(Q,Q1).

lt_transform(and(P,true), P1) :- !,
        lt_transform(P,P1).

lt_transform(and(P,Q), (P1,Q1)) :- !,
        lt_transform(P,P1),
        lt_transform(Q,Q1).

lt_transform(or(P,Q), or(P1, Q1)) :- !,
        lt_transform(P,P1),
        lt_transform(Q,Q1).

lt_transform(F,F).
/*
```

2.5 The Staggering Inefficiencies of Reasoning Disjunctively

Of course, a loop like:

```
not (not (possible_switch_setting(Ss,R), prove(Q,R,R)))
```

makes it painfully obvious that for any reasonable number of switches, the runtime will simply explode. If you are wondering why people find it so hard to imagine the world in a different way, to visualise world peace, this is the reason.

This problem is amelieorated by several things. First, although uniprocessor speed is no longer growing exponentially, total computer power per dollar *is* still growing exponentially–if you wonder what we'll be doing with 64-core microprocessors, there's your answer. But even today's processors can run thorugh all 2^{32} possibilities of a 32-bit vector in about a second or so. So its not hopeless.

Another strategy employed by natural languages is the use of *vague* predicates. Consider, even, the predicate "green," which could be true of many different shades. By bundling together various different colors together in one predicate, this allows us to reason about all the various ways in which, e.g. a block could be green without a seperate horn clause theory for each shade of green.

Still, as a general strategy, its best to stick with definite knowledge as much as possible. We'll call this the *Autistic Heuristic*.

Bibliography

[1] The Fracas Consortium, Robin Cooper, et al. Describing the Approaches The Fracas Project deliverable D8 Available online, just google for it... December 1994

[2] The Fracas Consortium, Robin Cooper, et al. The State of the Art in Computational Semantics: Evaluating the Descriptive Capabilities of Semantic Theories The Fracas Project deliverable D9 Available online, just google for it... December 1994

[3] William Ralph (Bill) Keller. Nested Cooper storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer, editors Natural Language parsing and Linguistic Theories pages 432-445 Reidel, Dordrecht, 1988.

[4] Larson, R.K. (1985). On the syntax of disjunction scope. Natural Language and Linguistic Theory 3:217, 265

[5] Patrick Blackburn and Johan Bos Computational Semantics for Natural Language Course Notes for NASSLLI 2003 Indiana University, June 17-21, 2003 Available on the web, just google for it....

[6] Mitchell P. Marcus, Beatrice Santorini and Mary ann Marciniewicz. Building a large annotated corpus of English: the Penn Treebank. in Computational Linguistics, vol. 19, 1993.

[7] Terrence Parsons Events in the semantics of English MIT press, Cambridge, Ma. 1990

[8] Fernando C.N. Pereira and Stuart M. Shieber Prolog and Natural Language analysis CSLI lecture notes no. 10 Stanford, CA 1987