The Sixth Chamber : Conjunction

Randall A. Helzerman

June 23, 2007

Contents

Chapter 1

Infrastructure for Sixth Chamber

```
*/

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

:-discontiguous(np/6).
:-discontiguous(defunkify/3).
:-discontiguous(case/2).
:-discontiguous(case/2).
:-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).
```

/*

1.3 Compling for query

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

1.4 Compiling to Horn Clauses

As usual, this is just an application of skeletons and techniques: the skeleton is the vanilla theorem prover, and the technique is to add another variable which is the list of atomic formlas:

```
*/
prove((A,B), R) :- prove(A, R), prove(B, R).

prove(A, R) :-
    clause((A :-B)),
    prove(B,R).

prove(A,_R) :- axiom(A).

prove(A,R) :- member(A,R).

/*
```

then we need some world knowledge

```
*/
% 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), w_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).
/*
```

1.4.1 Updating the Database

The first modification we have to make to the tranform clauses is that we don't want to immediately skolimze the existentially quantified variable to *constants* anymore. The reason is that a formula like:

```
exists(X,block(X))
```

might be talking about a block we already know about. So instead, we just want to accumulate the existentially quantified variables, and later try to match them up, if possible, with pre-existing objects.

So using the the c_transform clause as a skeleton, we add a difference list to hold the existentially quantified variables we encounter:

```
*/
c_transform(F,Cs,Vs) :- c_transform(F,Cs,[], Vs,[]).

c_transform(exists(X,R), CH,CT, [X|VH],VT) :- !,
        c_transform(R, CH,CT, VH,VT).

c_transform(and(R,S), CH,CT, VH,VT) :-!,
        c_transform(R, CH,CI, VH,VI),
        c_transform(S, CI,CT, VI,VT).

c_transform(true, CT,CT, V,V) :- !.
c_transform(F, [F|CT],CT, V,V).
/*
```

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

1.5 The Maxim of Minimum Mutilation

For each of the new existentially-quantified variables, we have to make a decision– either it refers either to an already-known object, or it refers to a new object. To make it easy to find out which, we'll change how we store the database. In addition to asserting axioms, we'll also assert two lists. One list will be a list of constants:

```
things([sk1, sk4, sk7,..]).
```

which are the items we've already postulated. Another list will be a list of the atomic formulas we hold true of those objects:

```
atoms([block(sk1), table(sk4), green(sk4), table(sk7), ...]).
```

So the idea is that when we get in the atomic formulas of the new sentence:

```
[block(X), green(X)]
```

we'll see if we can match up the variable X with any of the pre-existing objects without generating a contradictory database. If we can't, then we'll follow the Maxim of Thomas, and postulate a new object for X.

```
*/
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).
/*


*/

% Yet another instance of map.
generate_domains([], _Os,_Fs,_Db, []).
generate_domains([V|Vs], Os,Fs,Db, [domain(V,Pos)|Doms]) :-
     generate_domain(V, Os,Fs,Db, Pos),
     generate_domains(Vs, Os,Fs,Db, Doms).

% for variable V, this finds a list of objects in Os which
% V could be bound to without contradiction.
generate_domain(V, Os,Fs,Db, Pos) :-
     find_formulas_about_variable(Fs,V,Vfs),
     append(Vfs,Db,F1s),
     find_possible_values(Os,V,F1s, Pos).


% these two are very ugly routines which use
% meta-facilities of prolog to inspect terms and
% test variables
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).

% like member, but uses the meta-variable vacility
% in order to test for unified variables without
% unifying the variables.
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(Db2).

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

/*
```

1.6 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.

```
*/
assemble_formulas(Fs) :- findall(F,axiom(F),Fs).

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)
).


% transform to clauses, then assert.
process_logical_form(assertion,L) :-
c_transform(L,Fs,Vs),
assemble_formulas(Db1),
retractall(axiom(_)),
        extract_objects_from_formulas(Db1,Os),
mmm_update(Vs,Os,Fs,Db1,Db2),
assert_all_as_axioms(Db2).

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

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

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

1.7 Alpha and Beta Conversion Test Suite

Adompted from Blackburn and Bos, who adopted and extended from Milward.

```
*/

expression(1,P^(P*mia)*(X^walk(X)),
           walk(mia)).

expression(2,some(X,and(man(X),(P^some(X,and(woman(X),P*X)))*(Y^love(X,Y)))),
           some(X,and(man(X),some(Y,and(woman(Y),love(X,Y))))) ).

% Simple reduction
expression(3,(A^sleep(A))*mia,
           sleep(mia)).

% Reduction to another function:
expression(4,A^B^like(B,A)*mia,
           C^like(C,mia)).

% Reduction of inner expression leaving outer function:
expression(5,A^(B^like(B,A)*vincent),
           (C^like(vincent,C))).

% Reduction of inner expression leaving outer function,
% with var y shared.
expression(6,A^and(((B^like(B,A))*vincent),sleep(A)),
           (C^and(like(vincent,C),sleep(C)))).

% Reduction twice:
expression(7,A^B^like(B,A)*mia*vincent,
           like(vincent,mia)).

% Reductions in a nested position
```

```
expression(8,p((A^sleep(A))*vincent),
           p(sleep(vincent))).

% Reductions inside a variable predicate
expression(9,(A^(A*((B^sleep(B))*vincent))),
           (A^(A*sleep(vincent)))).

% No reductions possible
expression(10,(A^(A*sleep(vincent))),
           (A^(A*sleep(vincent)))).


% Nested reductions (apply to a function which must be applied again)
expression(11,((A^(A*vincent))*(B^sleep(B))),
           sleep(vincent)).

%
expression(12,((A^believe(mia,(A*vincent)))*(B^sleep(B))),
           believe(mia,sleep(vincent))).

% Copied functions
expression(13,((A^and((A*vincent),(A*mia)))*(B^sleep(B))),
           and(sleep(vincent),sleep(mia))).

expression(14,(((A^(B^and(((C^(C*(A*vincent)))*(D^probably(D))),
                         ((C^(C*(B*mia)))*(D^improbably(D))))))
               *(E^walk(E))) *
              (E^talk(E))),
           and(probably(walk(vincent)),improbably(talk(mia)))).

% Double application.
expression(15,((((A^(B^((C^((C*A)*B))*(D^(E^love(D,E))))))*jules)*mia),
           love(jules,mia)).

% Two functions with the same use of variable.
expression(16,
           (((A^(B^some(C,and((A*C),(B*C)))))
           *
           (D^boxer(D)))*(D^sleep(D))),
           some(C,and(boxer(C),sleep(C)))).

% Test for correctly dealing with the same variable name occurring twice
% (loops without alpha-conversion)
expression(17,(A^(_*A)) * (C^(A^like(A,C))),
           (_*(E^(F^like(F,E))))).

% Test for correctly performing alpha conversion
expression(18,((A^(B^(A*B)))*(C^(B^like(B,C)))),
           (D^(E^like(E,D)))).
```

```
% Test for correctly performing alpha conversion
expression(19,(((A^(B^((C^((C*A)*B))*(B^(A^love(B,A))))))*jules)*mia),
           love(jules,mia)).

% Test for correctly performing alpha conversion
expression(20,(A^(B^((C^((C*A)*B))*(B^(A^love(B,A)))))),
           (D^(E^love(D,E)))).

% Further alpha conversion testing
expression(21,(((A^(B^and(some(B,(A*B)),(A*B))))*(C^boxer(C)))*vincent),
           and(some(B,boxer(B)),boxer(vincent))).

% reduction of an application to an expression
expression(22,foo((X^X)*b)*a,foo(b)*a).

% reduction of an application to an expression--check to see that I've
% got reapply_stack recursion right
expression(23,foo((X^X)*b)*a*c*d*e,foo(b)*a*c*d*e).

/*
```

Chapter 2

The Sixth Chamber: Conjunction

What's one and one and one and one and one and one and one? –the red queen, in *Alice Through the Looking Glass*

2.1 Why we can't use unification

Jack and Jill went up the hill.

2.2 Lambda Calculus

2.3 Developing Alpha Conversion and Beta Conversion using Skeletons and Techniques

This parser is the skeleton we use:

2.3.1 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).

/*
```

2.3.2 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 "visitBoundVariables" 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).

/*
```

2.3.3 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).

/*
```

2.3.4 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).

/*
```

2.3.5 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.

/*
```

2.3.6 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).

/*
```

2.4 Grammar

2.4.1 Left Recursion

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

np(C,D*(X^and(A*X,N*X))) -->
{ C>0 },
det(D), adjective(A), noun(N).

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

     conjunction(CL),

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

2.4.2 Features and agreement

It sounds funnny to say:

```
The table and the block is green.
```

So the addition conjunction introduces number into the picture.

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

then verb phrases are apropriately modified:

```
vp(Number,P1,P2,L) --> verb(Number,P1,P2,_), adjective(L).

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

*/

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


% the block is green.
s(assertion,NPL*VPL) -->
    np(Number,2,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,2,NPL),

    vp(Number,[gap(W,Number)],[], VPL),
    ['?'].

% is there a block?
s(yn_question,NPL*VPL) -->
    verb(Number,[],[],W),
    advp([],[],AvpF),
    np(Number,2,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), adjective(A), noun(N).

np(plural, 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,_), adjective(L).

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


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

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


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


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

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

/*
```

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 lquantification 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