Project Report on Naproche Summer Internship

Bhoomija Ranjan
Mentor: Dr. Peter Koepke

July 17, 2008

Contents

1 The Goal

The broad goal of this project is to bridge the gap between natural language mathematics and Automated Theorem Provers(ATP) using mathematical and computer linguistic means. The aim is to construct a system which accepts a controlled but rich subset of ordinary mathematical language including TeX-style typeset formulas as input and transform them into formal statements, using linguistic techniques to extract relevant mathematical information about hypotheses and conclusion. Finally, this can be combined with a proof checking software to check the validity of the input statements.

2 Motivation for the Project

A joint initiative of Bernhard Schroeder(Linguistics, University of Duisburg-Essen) and Peter Koepke(Mathematical Institute, University of Bonn), NAPROCHE or the NAtural language PROof CHEcking, is a group dedicated to the formalisation of Natural Language. Previous work of the group includes the first formal proof of the Godel's completeness theorem in the established proof checking MIZAR. Now, we aim at bringing natural and formal mathematical languages closer by providing an interface to translate natural language into formal mathematical statements and checking their validity using Automated Theorem Provers. This project is projected to have many real world applications in industries such as aviation, telecom, etc. It would also help the layman to understand and use formal mathematics for his benefit.

3 The Project

3.1 The Group

- Bhoomija Ranjan
- Shruti Gupta
- Daniel Kuehlwein
- Doerthe Arndt

3.2 Our Work

3.2.1 Preliminaries: Learning Prolog

The first two weeks of the project comprised of learning Prolog. Prolog, an abbreviation for Programming in Logic is a logical programming language, and is much different from Java or C++, which are object-oriented programming languages. It is very useful as a medium for programming in a project like Naproche, where the functional and relational aspects of natural language acquire much significance, and have to be translated into formal statements without loss of information.
In this period, I also learnt about documentation in Prolog. It is an important aspect in day-to-day Prolog programming, as it makes the program more readable and understandable to the layman. I made a presentation for the same during the Naproche seminars on 23/05/08.

3.2.2 Programming

The basic structure of Naproche involves taking an input in the TEXMACs format (a Latex WYSIWYG editor), using Prolog and GULP(Graph Unification Logic Programming) to process the discourse into intermediate Proof Representation Structures(PRSs), converting these structures into standard TPTP Format for the ATPs, and finally returning the result from the ATPs as feedback to the system. In particular, we worked in the processing of PRSs to give input to the ATPs. In an earlier version, we did simple processing of PRSs, whereas in the later version (which is the current running version), we collectively enabled type handling by creating DOBSODs (an abbreviation for Doerthe, Bhoomija, Shruti and Daniel), which are GULP lists giving information about the input data as well as its type.
My individual contribution to the project was to translate DOBSODs into TPTP Format, feeding the input to the TPTP Prover, and feeding the result back into the system. I have written two programs *fof_check.pl* and *translation_tptp.pl* for the same, available on the subversion. They are described below. A detailed copy of the source codes is attached at the end of the report.

- **fof_check.pl**
  This program is used for checking whether a conjecture is provable from a given set of premises using standard TPTP Checkers, like Otter 3.3,etc. Presently the default prover is set as Otter 3.3 and is loaded from the file load.pl. The predicates used in the program are described as follows:
  Predicates

Predicate	Description
fof_check	The predicate takes in a list of premises in the form of DOBSODs as well as a list of formulae, and the id of the PRS in question.With these data, the premises and the formulae are passed to the specified TPTP Checker as 'axioms' and 'conjectures' and the result is written in three formats: a short version in the file 'final_output' in the user's working directory, the exact input to the prover in a file named ¡ inputPRS_ID ¿ and a detailed output version in the file with name of type ¡ outputPRS_ID ¿ in the folder Output, also present in the working directory.
check_theorem	This predicate checks whether the final_output has any string of the form "No Proof". If it doesn't, this implies the entire string of arguments have been proved.
prepare_premises	This predicate is used to convert a list of premises which are in the form of a list of DOBSODs into a list of axioms in the TPTP format.
prepare_formulae	This predicate is used to convert a list of formulae which are in the form of a list of DOBSODs into a list of conjecutures in the TPTP format.
write_to_file	This predicate writes the input into the specified file.
read_from_file	This predicate reads the specified string from a file stream OS.
check_word	This predicate returns the contents of the file as a list of ASCII Codes.
sublist	Given two sets X and Y, this predicate checks if X is a subset of Y.

- **translation_tptp.pl**
  This program translates a DOBSOD into an atom. The atoms so produced are then used in fof_check.pl to translate DOBSODs into the TPTP Format. The predicates used in the program are described as follows:

  Predicates

Predicate	Description
prepare_tptp	The predicate takes an input as DOBSOD and translates it into an a
prepare_list	This predicate takes in a list of DOBSODs and a list of bound variables as input, and processes them to give a list of atoms and the list of outbound variables.

3.2.3 Debugging and other tasks

In the later part of the project, I helped in debugging the earlier code involving the translation of discourse into PRSs and making the entire module work in a cohesive way.Some of the details are written below. A copy of the source codes I debugged is attached at the end of the report.

- **Ordinals Example**
  One of the most important tasks in debugging prs.pl was to create the PRS for the Ordinals Example. This meant changing the source code as well as the grammar in grammar.pl to a certain extent. One of the most important bugs removed was the 'Negation Bug', with the generic problem of equating negated PRSs of the form neg(PRS) with a normal PRS. Solving this bug allowed statements of the kind "assume that not X", "define (not) X iff (not) Y", "(not) X implies (not)Y", etc. to be generated into PRSs.

- **Changing Input Format**
  To decrease the running time of the code, it was necessary to change the input format of the MRefs in the PRSs from atoms to DOBSODs. This helped in decreasing the time for creating DOBSODs from PRSs. These changes were made to prs.pl, expr_grammar.pl, checker.pl and premises.pl, and their respective tests.

- **Accessibility in Negated PRSs**
  The accessibility in Negated PRSs was creating a problem as it was available to the PRSs below it, which was linguistically incorrect. Hence, changes were made to grammar.pl

- **Translation of DOBSODs into atoms**
  A separate code was required for translating DOBSODs into atoms for displaying PRSs. This

code is written in prs_export.pl

- **Allowing for multiple quantification** Statements like "for all x,y,..." were not getting parsed. Hence I helped making changes to grammar.pl.
- **Other Accessibility Issues**
  Certain accessibility issues like displaying "holds(DRef)" when a previously defined mathematical statement is repeated, as well as displaying previously defined variables and constants were solved by making certain changes to prs.pl. This helped in solving problems like accessibility in assumption closing triggers like "thus", etc.

- **Other Tasks**
  Other tasks included writing tests for prs.pl and checking the working of all the predicates in the program.

4 Acknowledgement

Naproche was a great learning experience for me. Apart from technical knowledge, it also taught me the benefits of team work. I would like to thank Prof. Koepke for giving this great opportunity to me to enhance my skills, Mr.Jip Veldman for all his help, and my colleagues for their unstinting support and guidance.

5 The Code

The source codes for the programs I have written and debugged are also attached below.

5.1 Source Code for fof_check.pl

```
fof_check(Premises,Formulae,PRS_ID) :-
check_time(Time),
check_prover(Checker),
check_size(Outputsize),
prepare_premises(Premises,Prepared_premises,0),
prepare_formulae(Formulae,Prepared_formulae,0),
        append(Prepared_premises,Prepared_formulae,Prepared_predicates),
atom_concat('Output/input',PRS_ID,Input_File),
write_to_file(Input_File,Prepared_predicates),
atom_concat('$NAPROCHE/Output/output',PRS_ID,File),
concat_atom(['perl $TPTP_HOME/SystemExecution/SystemOnTPTP -q',Outputsize,' ',
                    Checker,' ',Time,' ',Input_File,'>',File],Command),
shell(Command),
open(final_output,append,OS),
nl(OS),
write(OS,PRS_ID),
tab(OS,5),
write(OS,Prepared_formulae),
tab(OS,5),
( Outputsize=3 ->
( read_from_file(File,"Theorem") ->
     write(OS,'Theorem')
;
     write(OS,'No Proof')
  )
;
  (
   (Outputsize=0;Outputsize=1;Outputsize=2),!,
   name(Checker,Prover),
   append(Prover,"saysTheorem",Search_string),
     ( read_from_file(File,Search_string) ->
     write(OS,'Theorem')
;
     write(OS,'No Proof')
            )
  )
),
close(OS).

check_theorem :- catch(\+(read_from_file('$NAPROCHE/final_output',"NoProof")),
                        _,write('Nothing to check')).

prepare_premises(Premises,Prepared_premises,I) :-
Premises=[H1|T1],
!,
prepare_tptp(H1,H2,[],_),
        Newi is I+1,
        Prepared_premises=[Head|Tail],
        Head= fof(Newi,axiom,H2),
prepare_premises(T1,Tail,Newi).
prepare_premises([],[],_) :- !.

prepare_formulae(Formulae,Prepared_formulae,J) :-
        Formulae=[H2|T2],
!,
prepare_tptp(H2,H3,[],_),
Newj is J+1,
Prepared_formulae=[Head|Tail],
Head= fof(Newj,conjecture,H3),
prepare_formulae(T2,Tail,Newj).
prepare_formulae([],[],_) :- !.

write_to_file(File,Prepared_predicates) :-
Prepared_predicates=[Head|Tail],
!,
        open(File,append,OS),
write(OS,Head),
write(OS,.),
nl(OS),
close(OS),
        write_to_file(File,Tail).
write_to_file(_,[]) :- !.

read_from_file(File,Word) :-
atom_concat('$NAPROCHE/',File_in,File),
open(File_in,read,OS),
get0(OS,Char),
        check_word(Char,Charlist,OS),
sublist(Word,Charlist),
close(OS).

check_word(-1,[],_) :- !.
check_word(end_of_file,[],_) :- !.
check_word(Char,[Char|Chars],OS) :-
!,
get(OS,NextChar),
check_word(NextChar,Chars,OS).

prefix(X,Y) :- append(X,_,Y).
suffix(X,Y) :- append(_,X,Y).
sublist(X,Y):- suffix(S,Y),prefix(X,S).
```

5.2 Source Code for translation_tptp.pl

```
prepare_tptp(Grammar_exp,'$false',Bound_in,Bound_in):-
Grammar_exp = type~relation ..name~'$false',
!.
prepare_tptp(Grammar_exp,TPTP_exp,Bound_in,Bound_out):-
Grammar_exp = type~logical_symbol ..name~'~' ..args~[A],
!,
prepare_tptp(A,NewA,Bound_in,Bound_out),
concat_atom(['~','(',NewA,')'],TPTP_exp).
```

```
prepare_tptp(Grammar_exp,TPTP_exp,Bound_in,Bound_in):-
Grammar_exp = type~logical_symbol ..args~[A,B] ..name~X,
!,
prepare_tptp(A,NewA,Bound_in,_),
prepare_tptp(B,NewB,Bound_in,_),
concat_atom(['(',NewA,')',X,'(',NewB,')'],TPTP_exp).

prepare_tptp(Grammar_exp,TPTP_exp,Bound_in,Bound_in):-
Grammar_exp = type~relation ..args~Arglist ..name~X,
!,
prepare_list(Arglist,Newlist,Bound_in),
concat_atom([NewArglist,','],Newlist),
concat_atom([X,'(',NewArglist,')'],TPTP_exp).

prepare_tptp(Grammar_exp,TPTP_exp,Bound_in,Bound_in):-
Grammar_exp = type~quantifier ..args~[A,B] ..name~X,
!,
append(Bound_in,A,New_Bound_in),
prepare_list(A,NewA,New_Bound_in),
concat_atom([Newlist,','],NewA),
prepare_tptp(B,NewB,New_Bound_in,_),
concat_atom([X,'[',Newlist,']',':','(',NewB,')'],TPTP_exp).

prepare_tptp(Grammar_exp,TPTP_exp,Bound_in,Bound_in):-
Grammar_exp = type~variable ..name~Name,
!,
(member(Grammar_exp,Bound_in) ->
concat_atom(['V',Name],TPTP_exp)
;
concat_atom(['v',Name],TPTP_exp)
).

prepare_tptp(Grammar_exp,TPTP_exp,Bound_in,Bound_in):-
Grammar_exp = type~constant ..name~Name,
!,
concat_atom(['v',Name],TPTP_exp).

prepare_list([H|T],Atomlist,Bound_in):-
!,
prepare_tptp(H,NewH,Bound_in,_),
prepare_list(T,NewT,Bound_in),
concat_atom([NewH,',',NewT],Atomlist).
prepare_list([],'',_):- !.
```

5.3 The Ordinals Example

```
p(ord_example,Sentence) :-
Sentence = [
[assume, that, math("¬∃ x x ∈ ∅”)],
  [assume, that, for, all, math("x"),(,), not, math("x∈ x”)],
[define, math("Trans(x)"), if, and, only,if,math("∀ u∀ v (u ∈ v∧ v ∈ x → u ∈ x)”)],
[define, math("Ord(x)"), iff, math("Trans(x)∧ ∀ y(y ∈ x → Trans(y))”)],

[theorem],
[math("Ord(∅)”)],
[proof],
[consider, math("u∈ v”), and, math(”v ∈ ∅”)],
[then, math("u∈ ∅”)],
[math("∃ x x ∈ ∅”)],
[contradiction],
[then, math("u∈ ∅”)],
[thus, math("∀ u∀ v (u ∈ v ∧ v ∈ ∅ → u ∈ ∅)”)],
[hence, math("Trans(∅)”)],
[consider, math("y∈ ∅”)],
[then, math("∃ x x ∈ ∅)],
[contradiction],
[then, math("Trans(y)")],
[thus, math("∀ y(y ∈ ∅ → Trans(y))”)],
[hence, math("Trans(∅)”),and,math(”∀ y(y ∈ ∅ → Trans(y))”)],
[qed],

[theorem],
[for, all, math("x"), (,), math("y"), (,),math("x∈ y”),
and,math("y"), is, an, ordinal, implies, math("x"), is, an, ordinal],

[proof],

[consider, math("x∈ y”), and, math(”y”), is, an, ordinal],
[then, math("Ord(y)")],
[math("Trans(y)"), and, math("∀ z(z ∈ y → Trans(z))”)],
[in, particular, math("∀ z(z ∈ y → Trans(z))”)],
[observe, that, math("x∈ y”)],
[hence, math("Trans(x)")],
[consider, math("u∈ x”)],
[math("Trans(y)")],
[so, math("∀ u∀ v (u ∈ v ∧ v ∈ y → u ∈ y)”)],
[observe, that, math("u∈ x”), and, math(”x ∈ y”)],
[hence, math("u∈ y”)],
[recall, that, math("∀ z(z ∈ y → Trans(z))”)],
[hence, math("Trans(u)")],
[thus, math("∀ u(u ∈ x → Trans(u))”)],
[together, we, have, math("Trans(x)"), and,math("∀ u(u ∈ x → Trans(u))”)],
[hence, math([120]), is,an, ordinal],
[qed],

[theorem],
[there, is, no, math("x"), such, that, math("∀ u(u ∈ x ↔ Ord(u))”)],
[proof],
[assume,for,a,contradiction,that,there,is,an,math("x"),
such,that,math("∀ u(u ∈ x ↔ Ord(u))”)],
[lemma],
[math("Ord(x)")],
[proof],
[let, math("u∈ v”), and, math(”v ∈ x”)],
[then, math("u∈ v”)],
[math("v∈ x”)],
[math("Ord(v)")],
[together, we, have, math("u∈ v”), and, math(”Ord(v)”)],
[so, math("Ord(u)")],
[math("u∈ x”)],
[thus, math("∀ u∀ v (u ∈ v ∧ v ∈ x → u ∈ x)”)],
[hence, math("Trans(x)")],
[consider, math("y∈ x”)],
[then, math("Ord(y)")],
[math("Trans(y)∧∀ z(z ∈ y → Trans(z))”)],
[in, particular, math("Trans(y)")],
[thus, math("∀ y(y ∈ x → Trans(y))”)],
[together, we, have, math("Trans(x)∧∀ y(y ∈ x → Trans(y))”)],
[hence, math("x"), is, an, ordinal],
[qed],

[then, math("x∈ x”)],
[but, math("¬ x ∈ x”)],
[contradiction],
[thus, there, is, no, math("x"), such, that,
        math("∀ u(u ∈ x ↔ Ord(u))”)],
[qed]

].
```

Changes made in prs.pl to the predicates-

- **replace_prs**

```
replace_prs(Id, New, neg(In),neg(Out)) :-!,
        In = id~PId..
             drefs~Drefs..
             mrefs~Mrefs..
             conds~Conds..
             rrefs~Rrefs,
        Out = id~PId..
              drefs~Drefs..
              mrefs~Mrefs..
              conds~NewConds..
              rrefs~Rrefs,
        replace_prs_x(Id, New, Conds, NewConds).
```

- **attach_id**

```
attach_id(Id, Left ==> Right, NewLeft ==> Right) :-
        NewLeft = Left,
        ((NewLeft=neg(X)) ->(X= id~Id);(NewLeft = id~Id)).

attach_id(Id, Left => Right, NewLeft => Right) :-
        NewLeft = Left,
        ((NewLeft=neg(X)) ->(X= id~Id);(NewLeft = id~Id)).

attach_id(Id, Left := Right, NewLeft := Right) :-
        NewLeft = Left,
        ((NewLeft=neg(X)) ->(X= id~Id);(NewLeft = id~Id)).

attach_id(Id, neg(Statement), neg(NewStatement)) :-
        NewStatement = Statement,
        NewStatement = id~Id.

attach_id(Id, Statement, NewStatement) :-
        NewStatement = Statement,
        NewStatement = id~Id.
```

- **add_assumption**

```
add_assumption(A, In, Out) :-
        assumption_id(A, AId),
        new_index(E),
        concat_atom(['consec_', E], EmptyId),
        In = id_stack~[ActiveId|RestIds]..prs~PRS,
        Out = id_stack~[EmptyId, ActiveId|RestIds]..prs~NewPRS,
        get_prs_by_id(ActiveId, PRS, ActivePRS),
        ActivePRS = id~Id..
              drefs~Drefs..
              mrefs~Mrefs..
              conds~Conds..
              rrefs~Rrefs,
        Empty = id~EmptyId..
              drefs~[]..
              mrefs~[]..
              conds~[]..
              rrefs~[],
        NewActivePRS = id~Id..
              drefs~Drefs..
              mrefs~Mrefs..
              conds~[A => Empty|Conds]..
              rrefs~Rrefs,
        replace_prs(ActiveId, NewActivePRS, PRS, NewPRS),
        !, asserta(opens(AId, EmptyId)).

assumption_id(L ==> _, Id) :- L = id~Id.
assumption_id(L => _, Id) :- L = id~Id.
assumption_id(L, Id) :- L = id~Id.
assumption_id(neg(L),Id) :- L = id~Id.
```

- **add_definition**

```
add_definition(D1 := D2, In, Out) :-!,
        new_index(N),
        concat_atom(['definiens_', N], DefId),
        ((D2 = neg(X)) -> (X=id~DefId);(D2 =id~DefId)),
        In = id_stack~[ActiveId|RestIds]..prs~PRS,
        Out = id_stack~[ActiveId|RestIds]..prs~NewPRS,
        get_prs_by_id(ActiveId, PRS, ActivePRS),
        ActivePRS =  id~Id..
              drefs~Drefs..
              mrefs~Mrefs..
              conds~Conds..
              rrefs~Rrefs,
        NewActivePRS = id~Id..
              drefs~Drefs..
              mrefs~Mrefs..
              conds~[D1 := D2|Conds]..
```

```
                rrefs~Rrefs,
            replace_prs(ActiveId, NewActivePRS, PRS, NewPRS).
```

Changes made to grammar.pl-

```
implies --> {lexicon(X,implication)},X.

statement(S, Acc, NewAcc) -->
        {
                S = syn~(coord~yes..type~impl),
                S1 = syn~(coord~no..type~conj)
        },
        statement(S1, Acc, TmpAcc), implies, statement(S2, TmpAcc, NewAcc),
        {
                new_index(I1),
                new_index(I2),
                concat_atom(['antecedent_', I1], PremId),
                concat_atom(['succedent_', I2], ConclId),
                ((S1 = neg(X)) -> (X= id~PremId);(S1= id~PremId)),
                ((S2 = neg(Y)) -> (Y= id~ConclId);(S2=id~ConclId)),
                S = drefs~[]..mrefs~[]..conds~[S1 ==> S2]..rrefs~[]
        }.
```

5.4 Changing Input Format

I am attaching the main changes made to prs.pl, checker.pl, premises.pl below.

- **prs.pl**

```
add_expr(GlobalAccessibles, NewAccessibles, math(M), Out) :-
        free_vars(M, FreeVars,DOBSOD_M),
        list_to_set([math(DOBSOD_M)|FreeVars], Mrefs),
        drefs_and_conds(GlobalAccessibles, Mrefs, Drefs, Conds),
        Out = drefs~Drefs..
              mrefs~Mrefs..
              conds~Conds..
        rrefs~[],
        filter_local_accessibles(Conds, LocalAccessibles),
        union(GlobalAccessibles, LocalAccessibles, NewAccessibles).
```

- **checker.pl**

```
check_conditions(Id,[ holds(Y) |Rest],Mid_begin,Mid_end,
                Premises_begin,Premises_end,Check_trigger) :-
```

```
!,
% Find the Formula corresponding to Y.
% If it can’t be found in Mid_begin throw an error
Z = math_id(Y,Formula_PRS),
member(Z,Mid_begin),
% If FAIL THROW ERROR !!!
% Get rid of the math(..) part.
Formula_PRS = math(Formula_FOL),
(((Formula_FOL = type~quantifier),!);
 ((Formula_FOL = type~relation),!);
        ((Formula_FOL = type~logical_symbol),!)),!,
( Check_trigger = nocheck -> true
; fof_check(Premises_begin,[Formula_FOL],Id)
),
% If FAIL, THROW ERROR!!!
% Update premises
append(Premises_begin,[Formula_FOL],New_Premises_begin),
!,
check_conditions(Id,Rest,Mid_begin,Mid_end,
                    New_Premises_begin,Premises_end,Check_trigger).
```

- **premises.pl**

```
extract(Mrefs,Vars) :-
 extract(Mrefs,[],Vars).
extract([],X,X) :- !.
extract(Mrefs,Temp,Vars) :-
 Mrefs = [math(X)|T],
 append([X],Temp,Var1),
 extract(T,Var1,Vars).
```

5.5 Accessibility in Negated PRSs

```
negated --> {lexicon(X,negated)},X.

statement(neg(Statement), Acc, Acc) -->
        negated, statement(Statement, Acc, _).
```

5.6 Translation of DOBSODs into atoms

```
dobsod_to_atom(Grammar_exp,'$false'):-
        Grammar_exp = type~relation ..name~'$false',
        !.
```

```
dobsod_to_atom(Grammar_exp,Atom):-
        Grammar_exp = type~logical_symbol ..name~'~' ..args~[A],
        !,
        dobsod_to_atom(A,NewA),
        concat_atom(['~','(',NewA,')'],Atom).

dobsod_to_atom(Grammar_exp,Atom):-
        Grammar_exp = type~logical_symbol ..args~[A,B] ..name~X,
        !,
        dobsod_to_atom(A,NewA),
        dobsod_to_atom(B,NewB),
        concat_atom(['(',NewA,')',X,'(',NewB,')'],Atom).

dobsod_to_atom(Grammar_exp,Atom):-
        Grammar_exp = type~relation ..args~Arglist ..name~X,
        !,
        prepare_list(Arglist,Newlist),
        concat_atom([NewArglist,','],Newlist),
        concat_atom([X,'(',NewArglist,')'],Atom).

dobsod_to_atom(Grammar_exp,Atom):-
        Grammar_exp = type~quantifier ..args~[A,B] ..name~X,
        !,
        prepare_list(A,NewA),
        concat_atom([Newlist,','],NewA),
        dobsod_to_atom(B,NewB),
        concat_atom([X,'[',Newlist,']',':','(',NewB,')'],Atom).

dobsod_to_atom(Grammar_exp,Atom):-
        Grammar_exp = type~variable ..name~Atom,
        !.

dobsod_to_atom(Grammar_exp,Atom):-
        Grammar_exp = type~constant ..name~Atom,
        !.

prepare_list([H|T],Atomlist):-
        !,
        dobsod_to_atom(H,NewH),
        prepare_list(T,NewT),
        concat_atom([NewH,',',NewT],Atomlist).
prepare_list([],''):- !.
```

5.7 Allowing for multiple quantification

```
universal_quant --> {lexicon(X,universal_quant)},X.
statement(S, Acc, TmpAcc)  -->
        universal_quant, statement(LeftSide,Acc,TmpAcc),
                          [',''], statement_list(RightSide, TmpAcc, _),
        {
                new_index(I1),
                new_index(I2),
                concat_atom(['prefix_forall', I1], PrefixId),
                concat_atom(['matrix_', I2], MatrixId),
                attach_id(PrefixId, LeftSide, LeftSideWithId),
                attach_id(MatrixId, RightSide, RightSideWithId),
          S = drefs~[]..mrefs~[]..conds~[LeftSideWithId ==> RightSideWithId]
                      ..rrefs~[]
        }.

statement_list(LeftSide, Acc, TmpAcc) --> statement(LeftSide, Acc, TmpAcc).
statement_list(S,Acc,TmpAcc) --> statement_list(LeftSide,Acc,TmpAcc),
                                [','],statement(RightSide, TmpAcc,_),
                {
                new_index(I1),
                new_index(I2),
                concat_atom(['prefix_forall', I1], PrefixId),
                concat_atom(['matrix_', I2], MatrixId),
                attach_id(PrefixId, LeftSide, LeftSideWithId),
                attach_id(MatrixId, RightSide, RightSideWithId),
                S = drefs~[]..mrefs~[]..conds~[LeftSideWithId ==> RightSideWithId]
                            ..rrefs~[]
                }.
```

5.8 Other Accessibility Issues

```
drefs_and_conds(_, [], [], []).

drefs_and_conds(Acc, [math(Mref)|MRest], IRest, CRest) :-
        assoc_index(Acc, math(Mref), _, referenced),
(Mref=type~variable;
         Mref=type~constant),!,
        drefs_and_conds(Acc, MRest, IRest, CRest).

drefs_and_conds(Acc,[math(Mref)|MRest],[Index|IRest],
                [math_id(Index, math(Mref))|CRest]) :-
        (Mref=type~variable;
         Mref=type~constant),
        assoc_index(Acc, math(Mref), Index, novel),!,
        drefs_and_conds(Acc, MRest, IRest, CRest).

drefs_and_conds(Acc, [math(Mref)|MRest],[Index|IRest], [holds(Index)|CRest]) :-
        assoc_index(Acc, math(Mref), Index, referenced),
(Mref=type~relation;
         Mref=type~logical_symbol;
         Mref=type~quantifier),!,
        drefs_and_conds(Acc, MRest, IRest, CRest).

drefs_and_conds(Acc,[math(Mref)|MRest],[Index|IRest],[holds(Index),
                math_id(Index,math(Mref))|CRest]) :-
        (Mref=type~relation;
         Mref=type~logical_symbol;
         Mref=type~quantifier),
        assoc_index(Acc, math(Mref), Index, novel),!,
        drefs_and_conds(Acc, MRest, IRest, CRest).
```

References

[1] PatrickBlackburn, Johan Bos and Kristina Streignitz, "Learn Prolog Now!".

[2] SWI-Prolog site on Documentation,`http://www.swi-prolog.org/packages/pldoc.html`.