Naproche Summer Internship 2008

Shruti Gupta
Mentor: Dr. Peter Koepke

July 17, 2008

Contents

1 General Outline Of Naproche

Naproche - NAtural language PRoof CHEcker - aims at closing the gap between natural language mathematics and automated theorem provers. Using computer linguistic and mathematical means, we want to translate natural language mathematical texts into a format which is readable for automated theorem provers (ATP). Combining Texmacs, a Latex WYSIWYG editor, Naproche and an ATP gives a program which is able to check natural language mathematical texts for correctness.

1.1 Part of the project undertaken by our team

The Team :

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

We worked as a team to write a code in prolog which takes the input, converts it into the PROLOG format(the language being used by us), passes it through a prover and checks the validity and correctness of the proof.For doing this, we made use of Discourse Representation Theory and Proof Representation Structures mainly.

1.1.1 First version

The version we have now, is an improvement over of the one we made earlier about a month ago. The older version had the formulas in the form of strings. Though the version was almost complete, it could not implement a few cases properly and it was a little complex to fix the bugs because of the string format of formulas.

1.1.2 Current Version

The current version is a simpler, stabler and more efficient one. To make things simpler and quicker, we introduced a new format for the formulas, called 'DOBSOD' which is based on the PROLOG feature known as 'GULP'.

2 Work done by me

2.1 Reading work done

2.1.1 Learning PROLOG

During the first two weeks, we spent time in learning up PROLOG, the programming language we were supposed to use. For this, we used the book *'Learn Prolog Now'* by Patrick Blackburn,Johan Bos and Kristina Striegnitz; available on *http://www.coli.uni-sb.de/kris/learn-prolog-now*. We learnt about how to code in prolog and did the exercises given in the book for practice. We learnt about :

- Facts, Rules and Queries
- Matching and Proof Searching
- Recursion
- Lists
- Arithematic
- Grammars
- Cut and Negation
- Database Manipulation
- Working with files

2.1.2 Discourse Representation Theory and Proof Representation Structures

Discourse Representation Theory provides(DRT) a well-defined semantics for representing natural language discourse. DRT provides a basic data structure - the Discourse Representation Structure (DRS) which serves a two-fold purpose: it is used to represent content and to provide context. We referred to *'Generating Proof Representation Structures in the Project Naproche'* ,the master's thesis of Nickolay Mitov Kolev.
PRSs are a mode of generating a formal representation from informal mathematical discourse. So, we learnt about the structure of PRSs, in order to know how to create and read them. Discourse Representation Theory and Proof Representation Structures(PRS) are and the basis of the coding for our project.

2.1.3 GULP

GULP is a simple extension to Prolog that facilitates implementation of unification-based grammars by adding a notation for feature structures. For example, a:b..c:d denotes a feature structure in which a has the value b, c has the value d, and the values of all other features are unspecified. A modified Prolog interpreter translates feature structures into Prolog terms that unify in the desired way. Thus, the extension is purely syntactic.
We read up about GULP from Research Report AI-1994-06, *GULP 3.1: An Extension of Prolog Unification-Based Grammar* by Michael Covington,Artificial Intelligence Center, The University of Georgia. Gulp was an essential thing for us to learn because DOBSODs- the format for writing formulas is based on GULP. Also, GULP is essential for the implementation of DRSs and PRSs. Thus, GULP forms the basis of our coding style and ideas.

2.1.4 Unit testing

After the code is written, we need to test them as a Quality Assurance Measure. Tests are important to validate the final system. Also,it saves a lot of time if somebody returns to the application a few years later or needs to modify and debug it. Thus, it are important to document how the code is supposed to be used. Also, it saves time during development of big applications, and make it easier to trace bugs.
I read up about how we could use this feature of Prolog for the benefit of our project. I also gave a short presentation on the same in the *Formal Mathematik Seminar* held on 23rd May,2008. It included different features of Pl Unit testing, various ways to write tests, alongside discussing their advantages and disadvantages.

2.2 Code written

2.2.1 premises.pl

This module provides predicates to update the premises after some high level structure change in a prs. e.g. closing assumptions, new definition, negation etc. The code deals with the formulas written in a structured format which we call 'DOBSOD'. All these predicates get their input and are called from the module *checker.pl* which deals with them in their PRS form.

It contains 6 main predicates:

update_definitions	To add new definitions to the existing premises.
update_assumption	To add new assumptions to the existing premises.
negate_formulas	To negate a given list of formulas.
update_for_all	To add formulas with universal quantification to the existing premises.
update_there_exists	To add formulas with universal quantification to the existing premises.
update_implication	To add new implication formulas to the existing premises.

It also contains some subsidiary predicates required for internal use of the code:

make_conjunction	Conjuncts the given list of formulas using '&'.
freevar	Gives the list of free variables in the given formula.
extract	Extracts variables from a given list of Mrefs.
make_list	Gets the free variables from all the elements in the input list.

2.2.2 lexicon.pl

Lexicon lists all items and their properties which *grammar.pl* can parse.
grammar.pl is the grammar for the Naproche Language.

2.2.3 reverse_prs

This predicate returns the reversed form of the input PRS. It mainly involves reversing the order of the conditions of the PRS. Any PRS encountered as a condition will again be recursively reversed. It takes care of all the cases involved, like, negation,implication,assumption etc.

2.3 Testing

For each and every predicate written by me, I have written test files which contain test for each predicate written, no matter however small. The tests check for every possible case the predicate should be equipped to handle. I have written tests for the predicates written in *premises.pl* in the file *premises.plt* and also written some tests in the *checker.plt*.

2.4 Documentation

When working in a team, documentation is required for a consistent, shared and well-designed coding style. Even otherwise, documentation of the file or module as a whole, explains its design, purpose and relation to other modules. Therefore, we have commented well the code written by us, so that it is easy to read and understand.

2.5 Debugging

After everybody's codes were compiled together, some patching up work was required to get the Ordinals example running, as the code was not completely equipped to handle all the cases. Apart from the smaller bugs, a few major ones were :

- Negation case :
  - Problem :
    The code written initially was the inability to handle cases involving negated PRS's. This was due to the fact that a negated PRS is considered as a PRS technically, but this had not been taken into account while writing the code.
  - Solution :
    A new predicate *is_neg_prs* was written in the module *prs.pl*. This predicate checks if the input is a negated PRS. Thus, in the entire code wherever we were checking if the given input was a valid PRS before acting further on it, an additional clause for checking a valid negated PRS and further action on it was added. Similarly a new predicate *check_neg_prs* was also written in *checker.pl*.
- Existential Quantification and 'There is no' case :
  - Problem :
    Initially the PRS being formed for the Existential Quantification case and its negation were not compatible with the code written.
  - Solution :
    Changes were made in the file *grammar.pl* to form a correct PRS for quantification. Now, PRS involving existential quantification has the id 'prefix_there_exists'. PRS involving universal quantification has the id 'prefix_for_all'. The 'there is no' case is now treated as the negation of existential quantification.
- reverse_prs :
  The predicate *reverse_prs* was written to reverse the PRS at the end of the program before we display it. The need for this predicate arose to display the correct PRS, as the order of the conditions gets reversed earlier in the code while changing it from Discourse Representation Structure to a PRS.

3 Conclusion

The current version of Naproche that we have, is stable and runs correctly on the Ordinals example. As our plugin for Texmacs is not working at the moment, the example is hard coded in our source files.

3.1 Possible improvements

Modifications can be made in the code which allow the negation of a PRS to be treated as a PRS itself. This would demand a lot of further modifications in the entire program, but would eventually lead to a code which is much simpler, easier to handle.

A Predicates in *premises.pl*

```
%% update_definitions(+Premises:list(DOBSOD),-New_Premises:list(DOBSOD),
                      +List_A:list(DOBSOD),+List_B:list(DOBSOD)).

update_definitions(Premises,Premises,[],_) :- !.
update_definitions(Premises,Premises,_,[]) :- !.

update_definitions(Premises,New_Premises,List_A,List_B) :-
 List_B = [H|T],
 make_conjunction(List_A,A),
 New = type~logical_symbol ..name~'<=>' ..arity~2 ..args~[A,H],!,
 freevar(New,Freevars),
 Newdef = type~quantifier ..name~'!' ..arity~2 ..args~
           [Freevars,type~logical_symbol ..name~'<=>' ..arity~2 ..args~[A,H]],!,
 append(Premises,[Newdef],Temp),
 update_definitions(Temp,New_Premises,List_A,T).


%% update_assumption(+Premises:list(DOBSOD),-New_Premises:list(DOBSOD),
                     +List_A:list(DOBSOD),+List_B:list(DOBSOD)).

update_assumption(Premises,Premises,_,[]) :- !.
update_assumption(Premises,Premises,[],_) :- !.

update_assumption(Premises,New_premises,List_A,List_B) :-
 List_B = [H|T],
 make_conjunction(List_A,A),
 New = type~logical_symbol ..name~'=>' ..arity~2 ..args~[A,H],!,
 freevar(New,Freevars),
 ( Freevars = [] ->
   Newassump = New;
    ( Newassump = type~quantifier ..name~'!' ..arity~2 ..args~
```

```
                  [Freevars,type~logical_symbol ..name~'=>' ..arity~2 ..args~[A,H]]
    )
  ),!,
 append(Premises,[Newassump],Temp),
 update_assumption(Temp,New_premises,List_A,T).


%% make_conjunction(+In:list(DOBSOD),-Out:DOBSOD)

make_conjunction([],[]) :- !.
make_conjunction([X],X) :- !.

make_conjunction(In,Out) :-
 In = [H|T],
 make_conjunction(T,Temp),
 Out = type~logical_symbol ..name~'&' ..arity~2 ..args~[H,Temp].


%% negate_formulas(+Formulas:list(DOBSOD),-Neg_formulas:list(DOBSOD)).

negate_formulas([],X,X) :- !.

negate_formulas(Formulas,Temp,Neg_formulas) :-
 Formulas = [H|T],
 New = type~logical_symbol.. name~'~' ..arity~1 .. args~[H],
 append(Temp,[New],Temp1),
 negate_formulas(T,Temp1,Neg_formulas).

negate_formulas(Formulas,Neg_formulas) :-
 negate_formulas(Formulas,[],Neg_formulas).


%% update_for_all(+Premises:list(DOBSOD),-New_premises:list(DOBSOD),
                  +Mrefs:list(atom),+Formulas:list(DOBSOD)).

update_for_all(Premises,Premises,_,[]) :- !.
update_for_all(Premises,Premises,[],_) :- !.

update_for_all(Premises,New_premises,Mrefs,Formulas):-
 Formulas = [H|T],
 extract(Mrefs,Vars),
 New= type~quantifier ..name~ '!' ..arity~2 ..args~[Vars,H],!,
 append(Premises,[New],Temp),
 update_for_all(Temp,New_premises,Mrefs,T).
```

```
%% update_there_exists(+Premises:list(DOBSOD),-New_premises:list(DOBSOD),
                       +Mrefs:list(atom),+Formulas:list(DOBSOD)).

update_there_exists(Premises,Premises,_,[]) :- !.
update_there_exists(Premises,Premises,[],_) :- !.

update_there_exists(Premises,New_premises,Mrefs,Formulas):-
 Formulas = [H|T],
 extract(Mrefs,Vars),
 New= type~quantifier ..name~ '?' ..arity~2 ..args~[Vars,H],!,
 append(Premises,[New],Temp),
 update_there_exists(Temp,New_premises,Mrefs,T).


%% update_implication(Premises:list(DOBSOD),New_premises:list(list(DOBSOD)),
                      Formula_A:list(DOBSOD),Formula_B:list(DOBSOD)).

update_implication(Premises,New_premises,Formula_A,Formula_B) :-
 update_assumption(Premises,New_premises,Formula_A,Formula_B).


%% freevar(+A:DOBSOD, -Free_A:list(DOBSOD)).

freevar(A,Free_A) :-
 A = type~constant,
 Free_A = [],!.

freevar(A,Free_A) :-
 A = type~variable,
 Free_A=[A],!.

freevar(A,Free_A) :-
 A = type~function..args~ARGS,
 make_list(ARGS,Free_A),!.

freevar(A,Free_A) :-
 A = type~relation..args~ARGS,
 make_list(ARGS,Free_A),!.

freevar(A,Free_A) :-
 A = type~logical_symbol..args~ARGS,
 make_list(ARGS,Free_A),!.

freevar(A,Free_A) :-
 A= type~quantifier..args~[Bound_list,Formula],
 make_list(Bound_list,Bound),
```

```
 freevar(Formula,Free),
 subtract(Free,Bound,Free_A),!.


%% make_list(+List:list(DOBSOD),-Output:list(DOBSOD))

make_list([],[]).

make_list([X|Rest],Output) :-
 freevar(X,Free_X),
 make_list(Rest,Tmp),
 union(Free_X,Tmp,Output),!.


%% extract(+Mrefs:list(atom),-Vars:list(int)).

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).
```

B lexicon.pl

```
% Negated statement
lexicon([not],negated).

% Universal quantification
lexicon([for,all],universal_quant).

% Existential Quantification
lexicon([there,exists],existential_quant).
lexicon([there,is],existential_quant).

% Implication
lexicon([implies],implication).

% Iff
lexicon([iff],iff).
```

```
lexicon([if,and,only,if],iff).

% Conjunction
lexicon([',',],conj).
lexicon([&],conj).

% Structural markers
lexicon([qed],struct_marker).
lexicon([theorem],struct_marker).
lexicon([proof],struct_marker).
lexicon([lemma],struct_marker).

% Statement prefixes
lexicon([then],statement_prefix).
lexicon([hence],statement_prefix).
lexicon([recall, that],statement_prefix).
lexicon([but],statement_prefix).
lexicon([in, particular],statement_prefix).
lexicon([observe, that],statement_prefix).
lexicon([together, we, have],statement_prefix).
lexicon([so],statement_prefix).


% Definition Trigger
lexicon([define],def_trigger).

% Assumption Trigger
lexicon([assume, that],assumption_trigger).
lexicon([consider],assumption_trigger).
lexicon([let],assumption_trigger).
lexicon([assume, for, a, contradiction, that],assumption_trigger).

% Closing Trigger
lexicon([thus],closing_trigger).

% Contradiction
lexicon([contradiction],contradiction).

% Noun phrases
lexicon([is],verb).
lexicon([be],verb).
lexicon([a],determiner).
lexicon([an],determiner).
lexicon([ordinal],noun).
```

C predicate reverse_prs

```
%% reverse_prs(PRS:+PRS, RevPRS:-PRS).

reverse_prs(PRS,ReversePRS) :-
 (PRS = id~Id ..conds~Conds ..drefs~Drefs ..mrefs~Mrefs ..rrefs~Rrefs ->
        (reverseConds(Conds,ReverseConds),
         ReversePRS = id~Id ..conds~ReverseConds ..drefs~Drefs ..mrefs~Mrefs ..rrefs~Rrefs
        );
        true
 ),
 (PRS = neg(X) ->
  X = id~Id ..conds~Conds ..drefs~Drefs ..mrefs~Mrefs ..rrefs~Rrefs ->
        (reverseConds(Conds,ReverseConds),
         ReversePRS = neg(id~Id ..conds~ReverseConds ..drefs~Drefs ..mrefs~Mrefs ..rrefs~Rrefs)
        );
        true
 ).

 %% reverseConds(+Conds,-ReverseConds).
% reverses the order of conditions in a PRS.

reverseConds(Conds,ReverseConds) :-
 accrev(Conds,[],ReverseConds).

% Conditions is empty set
accrev([],A,A) :- !.

% Condition is a PRS
accrev([H|T],Temp,R) :-
  accrev(T,[RevH|Temp],R),
  (is_prs(H) -> reverse_prs(H,RevH)),!.

% Condition is a definition
accrev([H|T],Temp,R) :-
 accrev(T,[RevH|Temp],R),
   (H = (A := B) ->
   reverse_prs(A,RevA),
   reverse_prs(B,RevB),
   RevH = (RevA := RevB)
   ),
  !.

% Condition is an assumption
accrev([H|T],Temp,R) :-
```

```
 accrev(T,[RevH|Temp],R),
   (H = (A => B) ->
   reverse_prs(A,RevA),
   reverse_prs(B,RevB),
   RevH = (RevA => RevB)
   ),
  !.

% Condition is an implication
accrev([H|T],Temp,R) :-
 accrev(T,[RevH|Temp],R),
   (H = (A ==> B) ->
   reverse_prs(A,RevA),
   reverse_prs(B,RevB),
   RevH = (RevA ==> RevB)
   ),
  !.

% Condition is a negation
accrev([H|T],Temp,R) :-
 accrev(T,[RevH|Temp],R),
   (H = neg(X) ->
     reverse_prs(X,RevX),
    RevH = neg(RevX)
   ),
   !.

% Condition is none of the above, then the condition is left as it is
accrev([H|T],Temp,R) :-
 accrev(T,[RevH|Temp],R),
 RevH=H.
```

D Debugging

.1 Modifications to handle negations

```
is_neg_prs(What) :-
   What = neg(X),
   is_prs(X).

PRS_A := PRS_B :-
        (is_prs(PRS_A);is_neg_prs(PRS_A)),
        (is_prs(PRS_B);is_neg_prs(PRS_B)).
```

```
PRS_A => PRS_B :-
        (is_prs(PRS_A);is_neg_prs(PRS_A)),
        (is_prs(PRS_B);is_neg_prs(PRS_B)).

PRS_A ==> PRS_B :-
        (is_prs(PRS_A);is_neg_prs(PRS_A)),
        (is_prs(PRS_B);is_neg_prs(PRS_B)).
```

Similar changes were made in *checker.pl*.

.2 Modifications to handle existential quantification and its negation

```
% an existentially quantified statement
existential_quant --> {lexicon(X,existential_quant)},X.
statement(S, Acc, TmpAcc)  -->
 existential_quant,det,statement(LeftSide,Acc,TmpAcc),[such,that],statement(RightSide,TmpAcc,_),
  {
   new_index(I1),
   new_index(I2),
   concat_atom(['prefix_thereexists', I1], PrefixId),
   concat_atom(['matrix_', I2], MatrixId),
   attach_id(PrefixId, LeftSide, LeftSideWithId),
   attach_id(MatrixId, RightSide, RightSideWithId),
   S = drefs~[]..mrefs~[]..conds~[LeftSideWithId ==> RightSideWithId]..rrefs~[]
  }.

% an existentially quantified statement
existential_quant --> {lexicon(X,existential_quant)},X.
statement(S, Acc, TmpAcc)  -->
 existential_quant,statement(LeftSide,Acc,TmpAcc),[such,that],statement(RightSide, TmpAcc,_),
  {
   new_index(I1),
   new_index(I2),
   concat_atom(['prefix_thereexists', I1], PrefixId),
   concat_atom(['matrix_', I2], MatrixId),
   attach_id(PrefixId, LeftSide, LeftSideWithId),
   attach_id(MatrixId, RightSide, RightSideWithId),
   S = drefs~[]..mrefs~[]..conds~[LeftSideWithId ==> RightSideWithId]..rrefs~[]
  }.


% 'There is no' statement
statement(neg(S), Acc, TmpAcc)  -->
existential_quant,[no],statement(LeftSide,Acc,TmpAcc),[such,that],statement(RightSide, TmpAcc,_),
```

```
{
 new_index(I1),
 new_index(I2),
 concat_atom(['prefix_thereexists', I1], PrefixId),
 concat_atom(['matrix_', I2], MatrixId),
 attach_id(PrefixId, LeftSide, LeftSideWithId),
 attach_id(MatrixId, RightSide, RightSideWithId),
 S = drefs~[]..mrefs~[]..conds~[LeftSideWithId ==> RightSideWithId]..rrefs~[]
}.
```