Internship Report

Mona Rahn

Mathematical Institute, University of Bonn
mona.rahn@uni-bonn.de
http://www.naproche.net

1 The Naproche Project

The Naproche (*Natural Language Proof Checking*) project is a joint initiative of Peter Koepke and Bernhard Schröder. The aim is, firstly, to construct a system which accepts a controlled subset of ordinary mathematical language and to transform this language into formal statements. Secondly, we want to convert the statements into first-order logic so that a automated theorem prover (ATP) can check them.

2 An Overview over my Work

Figure 1 shows the architecture of the Naproche system. The boxes represent the major modules. Next to the downward arrows, the output format of the corresponding module is shown. The files mentioned explicitly in the boxes are the ones I mainly worked with.

The procedure is the following: At first, the user gives the input, supported by the file *help.html*. This is processed by *input_utils.pl* into a Naproche readable format so that a PRS can be created which is checked by *checker.pl* - my main working field. During these steps, error and warning messages provide a feedback to the user. If the input can be proven, the output is "Theorem".

3 The First Steps

In the first week, I familiarized myself with Prolog and the concept of *Proof Representation Structures* (PRS, see section 7.2).
At the beginning, I read the introduction of Blackburn and Bos into Representation Structures [1]; the exercises given in the two books helped me to get a better grasp of the underlying principles. These books cover *Discourse Representation Structures* (DRS), which is a technique computational linguists developed to automatically extract the semantics of a natural language text. The PRS used in Naproche are a modification of these structures.

In the first week I also studied Prolog, the programming language in which Naproche is written. Since I had never programmed in a declarative programming language before but only in imperative languages like C, Prolog appeared unusual

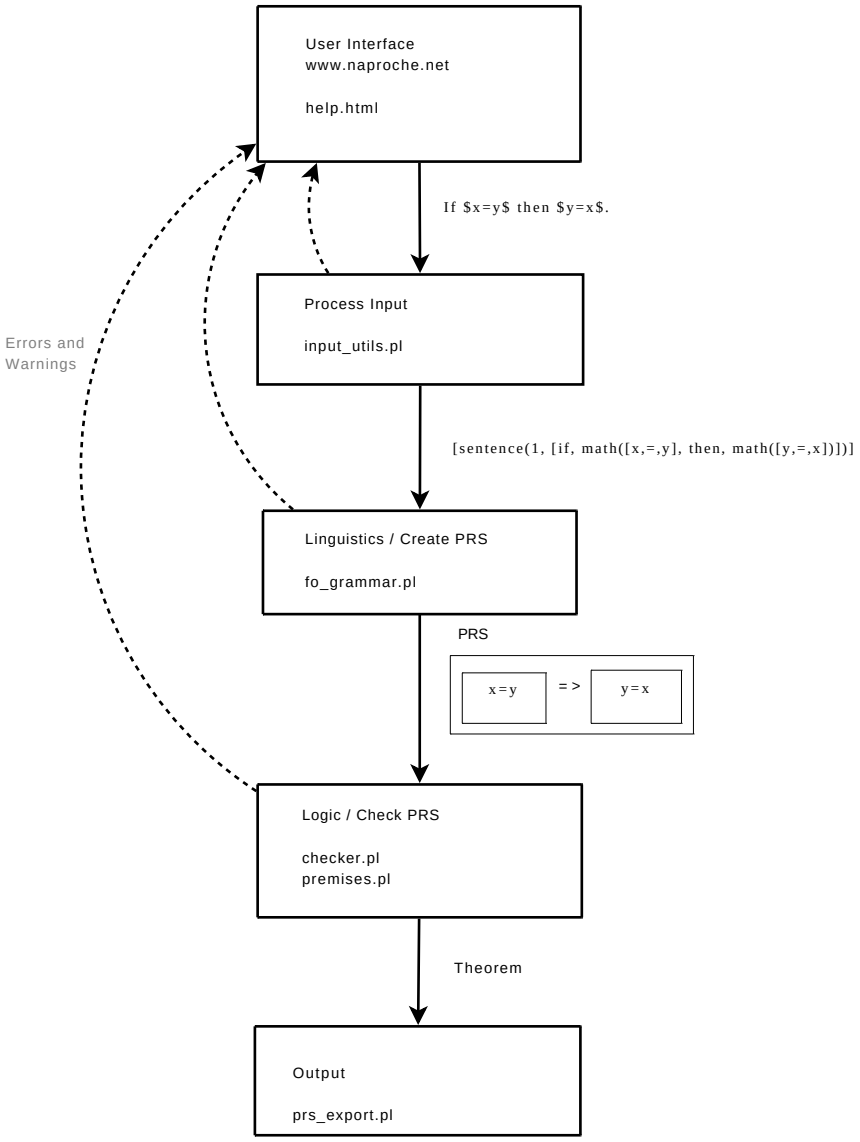

Fig. 1. The architecture of the Naproche system.

to me at first because there are no commands like "while"; but after I had written my first predicates in Prolog, I quickly got used to it. The website by Blackburn and Bos, Learn Prolog Now[1], was a really helpful (and entertaining) introduction. In this period, I also learnt how to implement tests in Prolog, which is an incredibly useful tool for finding errors. During my internship, I wrote tests for every predicate I programmed and more than thirty for *checker.pl* alone. Furthermore, I learnt how to write the documentation for my programs in the pldoc style and how to use the Prolog command `trace` for debugging.

4 The Module "input_utils.pl"

My first two programs written in Prolog were `word` and `input_math`. Both are part of the file *input_utils.pl* and used by the predicate `create_naproche_input`, which is responsible for processing the input into a form from which the PRS can be built. The procedure is the following:
Naproche gets the input in the form of a single atom, e.g.

```
'There is no $y$ such that $y \in \emptyset$.'
```

Then the atom is converted into a list of characters from which sentences are built. A sentence has an ID and is a list of words and mathematical input followed by a terminal symbol. In our example, the output sentence is

```
[
sentence(1, [there, is, no, math([y]), such, that,
    math([y, \u2208, \u2205])])
]
```

In order to build the sentence, my programs are used. `word` recursively defines a word; similarly, `input_math` processes mathematical input (enclosed by $ signs). `input_math` can also handle some LaTeX commands beginning with a backslash like `\in` and `\emptyset` in our example. For this feature, I wrote the predicate `latex_lexicon` which converts LaTeX into Unicode. It can also parse more complex LaTeX commands like `\sqrt{x}` or `\frac{\sqrt{x}}{y}`. A list of commands can be found on the Naproche website.

5 An Introduction into Error Messages

Syntactically the example given above is correct. For the case that the input is not correct, I have written error messages as a feedback to the user. In *input_utils.pl*, examples are:

[1] http://www.learnprolognow.org/

Input	Error Messages
`hallo`	message(error, inputError, create_naproche_input, hallo, Could not parse input.)
	message(error, inputError, sentence, 0, No terminal symbol found.)
`$x \a y$.`	message(error, inputError, sentence, $ x \a y$, Could not parse math mode.)
	message(error, inputError, input_math, \a, Latex command not supported.)

The error messages always have the same form:

- Either *error* or *warning* as the first entry; after an error, the program fails
- A specification of the error type, like *inputError*, *prsError* or *checkError*
- The predicate which throws the error message
- The wrong code or the ID of the wrong PRS (or 0 if the whole input is concerned)
- A description of the error

6 The Module "fo_grammar.pl"

The module *fo_grammar.pl* is mainly used in the PRS construction in order to process the mathematical input. It takes the mathematical output of `input_math` (see section 4) and converts it into DOBSOD format while extracting the free variables. For example, the input

```
[f,(,x,),=,y]
```

is transformed into

```
type∼function..name∼f..args∼[
     type∼variable..arity∼0..name∼x..args∼[]]
```

and the free variables are

```
[type∼variable..arity∼0..name∼x..args∼[]].
```

To maintain the order of the variables in the DOBSOD list, I wrote the predicate `union_in_right_order` which unifies two lists in the right order.

I also programmed the predicate `fo_chained_formula` so that chained formulas like can be parsed. For example, the input

```
[x,<,y,<,z]
```

is transformed into

```
type∼logical_symbol..name∼&..arity∼2..args∼[
     type∼relation..name∼less..args∼[
          type∼variable..name∼x..arity∼0..args∼[],
          type∼variable..name∼y..arity∼0..args∼[]
     ],
     type∼relation..name∼less..args∼[
```

```
                type~variable..name~y..arity~0..args~[],
                type~variable..name~z..arity~0..args~[]]
]
```

which is DOBSOD notation for "x<y and y<z".

Furthermore, I implemented the error messages in *fo_grammar.pl*. An example is the message

```
message(syntax,fo_term, X,'Term expected')
```

where X is the concatenation of the first five atoms from the place where the term is expected.

7 The Module "checker.pl"

7.1 Overview

My task in the internship primarily involved collaborating in *checker.pl*. This program mainly consists of two predicates: `check_prs` and `check_conditions`.

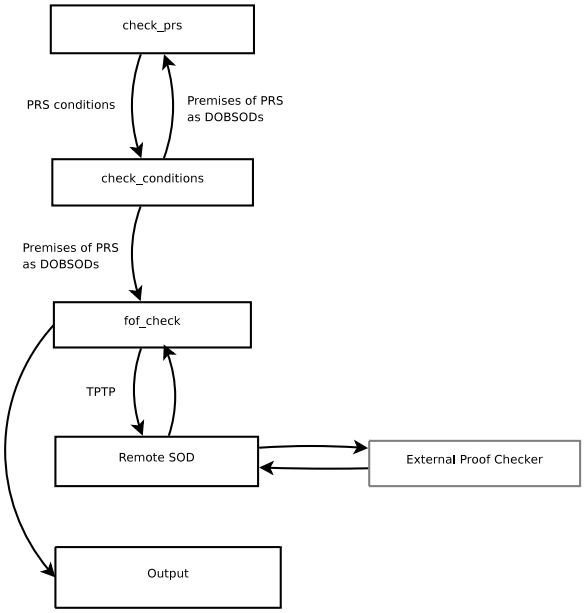

Fig. 2. Flowchart of checker.pl

The general procedure is the following: At first, `check_prs` gets the finished PRS as an input. As shown in figure 2, each of the conditions of the PRS is pro-

cessed by `check_conditions`; if the check-trigger is set to "check", the predicate `fof_check` checks it using an external ATP.

The PRS is logically valid if each of its conditions is valid.

7.2 The Input: Proof Representation Structures

The input of `check_prs` is a *Proof Representation Structure*. A PRS has the following constituents:

- The Identification number (*id*)
- A list of mathematical referents (*mrefs*)
- A list of discourse referents (*drefs*)
- A list of textual referents (*rrefs*)
- An ordered list of conditions (*conds*)
- A list of the discourse and mathematical referents accessible at the beginning of the PRS (*accbefore*)
- A list of the discourse and mathematical referents accessible at the end of the PRS which consists of the list *accbefore* and the new discourse and mathematical referents (*accafter*)

A simple example PRS (the PRS corresponding to the input "x=y") is:

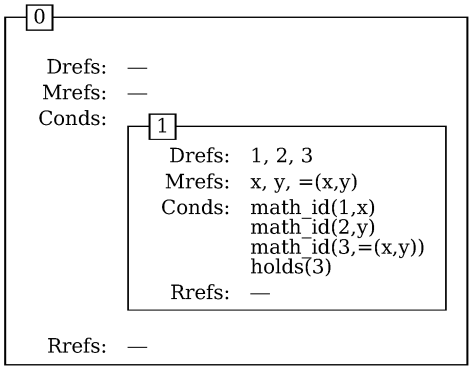

Fig. 3. An example PRS

A PRS can have the following conditions:

- for any n-ary predicate p $predicate(X_1, \ldots, X_n, p)$
- $holds(X)$, representing the claim that the formula referenced by X is true.
- $math\_id(X, Y)$, which binds a discourse referent X to a mathematical referent Y (a formula or a term).
- $PrsA$

- $neg(PrsA)$, representing a negation.
- $PrsA := PrsB$, representing a definition.
- $PrsA \Longrightarrow PrsB$, representing an assumption A and the set of claims B made inside the scope of this assumption.
- $PrsA \Rightarrow PrsB$, representing an implication A and the set of claims B made inside the scope of this implication. If $PrsA$ is empty, the condition represents a universally quantified formula
- $PrsA \Leftarrow PrsB$, representing a reversed implication
- *contradiction*, representing a contradiction.
- $PrsA \Leftrightarrow PrsB$, representing an equivalence
- $\_ :: PrsA \Rightarrow PrsB$, representing an function definition
- $PrsA \vee PrsB$, representing a disjunction
- $>< [PrsA, PrsB, \ldots]$, representing an exclusive disjunction

7.3 The Predicate "check_prs"

The predicate `check_prs` has five arguments:

+ PRS:prs is the input PRS
+ PremisesBegin:list(DOBSOD) is the list of premises that already have been parsed, given in DOBSOD format[2], which is a formula or a term stored as a tree.
+ CheckTrigger:(check / nocheck) gives us the option to either try to prove every formula we encounter (check) or to just go through the structure of the PRS parsing every formula without running a prover (nocheck).
- PremisesEnd:list(DOBSOD) is PremisesBegin with the formula image of the current PRS appended

Depending on the PRS, `check_prs` proceeds differently. We distinguish between four cases (not counting the error cases):

1. The PRS is a structure PRS, i.e. a theorem or a lemma (indicated by "theorem" or "lemma" in the PRS id)
2. The PRS is an induction PRS
3. The PRS is neither a structure PRS nor an induction PRS and has no variables in its *mrefs*
4. As case three, only that it has variables in its *mrefs*

The induction PRS case and case four were implemented by me; in the other two, I fixed some bugs and improved the code. An example for an induction in the Naproche Controlled Natural Language is

```
By induction, for all $x$ $ord(x)$.
```

The corresponding PRS is shown in figure 4. Note the word "induction" in the *rrefs* of the PRS. 4.

[2] For more information about this format, see the internship report of Bhoomija Ranjan, `http://www.math.uni-bonn.de/people/logic/publications/MT-2008-02.pdf`

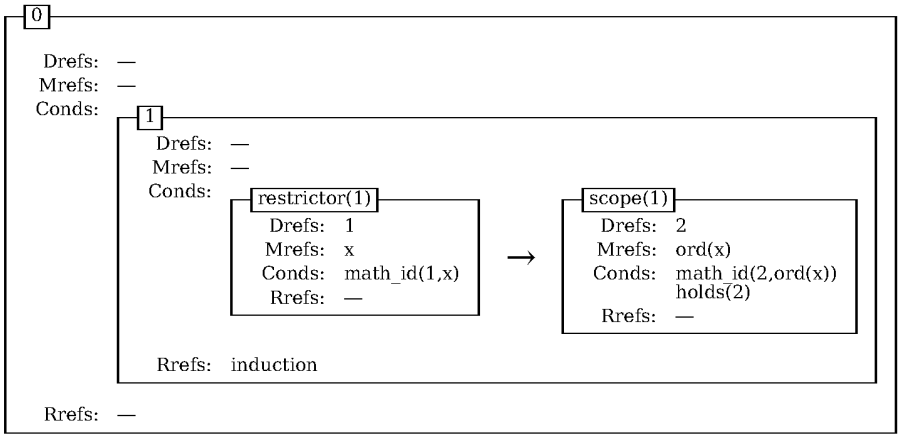

Fig. 4. An induction PRS

At first, `check_prs` just extracts the formula image of the PRS (using the predicate `check_conditions` which I will describe later). In our example, this would be:

```
type~quantifier..name~!..arity~2..args~[
    [x],
    type~relation..name~ord..arity~1..args~[
        type~variable..name~x..arity~0,
    ]
]
```

which is DOBSOD notation for "For all x: ord(x)". Now we check if the formula image is one universally quantified formula; if this is not the case, the error message

```
message[logic,check_prs,AtomId,
        Wrong formula for induction PRS
        (must be universally quantified).]
```

is thrown and `check_prs` fails. In our case, the input is correct. In order to prove a statement by induction, one has to prove the base case and the inductive step. Therefore, we proceed with making the induction formulas using the predicate `make_induction_formulas`, which can be found in *premises.pl*. In our example, it produces the following formula for the the base case:

```
type~relation..name~ord..arity~1..args~[
```

```
type∼constant..name∼1..arity∼0,
]
```

and for the inductive step:

```
type∼quantifier..name∼!..arity∼2..args∼[
    [x],
    type∼logical_symbol..name∼(=>)..arity∼2..args∼[
    type∼relation..name∼ord..arity∼1..args∼[
        type∼variable..name∼x..arity∼0,
        ]
    ]]
```

If CheckTrigger is set to "check", an ATP now tries to derive these formulas from PremisesBegin. PremisesEnd is instantiated with the concatenation of PremisesBegin and the universally quantified formula "for all x: ord(x)".

7.4 The Predicate "check_conditions"

The predicate `check_conditions` has six arguments:

+ Id:atom is the Id of the PRS that is currently checked
+ Accafter:list(DOBSOD) is the list of Math-IDs of the PRS
+ Conditions:list(DOBSOD) is the list of conditions which we try to prove
+ PremisesBegin:list(DOBSOD) is the list of premises from which we try to prove the PRS
- PremisesEnd:list(DOBSOD) is PremisesBegin with the formulae of Conditions appended
+ CheckTrigger:(check / nocheck) indicates whether the formulae should be checked by an ATP

`check_conditions` processes the conditions of a PRS sequentially and proceeds differently depending on the condition type. I implemented the majority of the different cases in *checker.pl* and updated the rest. Two examples are an exclusive disjunction and an assumption.

An **exclusive disjunction** condition $>< [PrsA, PrsB, \ldots]$ is created when the user writes "Precisely one of the following cases holds: ...". `check_conditions` can process up to four disjuncted PRSs. The case when we have two is treated as follows:

At first, we calculate the formula image of PrsA and PrsB by using `check_prs` with CheckTrigger = "nocheck". Then we quantify the premises of PrsA existentially over the variables in the *mrefs* of PrsA and proceed similarly with PrsB; let us call the results QPremisesA and QPremisesB (Q stands for quantified). The formulae corresponding to the exclusive disjunction

$QPremisesA \vee QPremisesB$
$QPremisesA \Rightarrow \neg QPremisesB$
$QPremisesB \Rightarrow \neg QPremisesA$

are constructed; then an ATP checks then if CheckTrigger is set to "check". Then the variable NewPremisesBegin is instantiated with the concatenation of PremisesBegin and these formulae and

```
check\_conditions(Id, Accafter, Rest,
                  NewPremisesBegin, PremisesEnd, CheckTrigger)
```

is called for the rest of the conditions. Note that NewPremisesBegin is the new PremisesBegin so that the formulae just parsed are treated like premises. PremisesEnd is just passed to the next call of `check_conditions`; it is instantiated with PremisesBegin when all conditions have been parsed and therefore the list Conditions is empty.

An **assumption** condition $PrsA \implies PrsB$ is built when the input is "Fix...", "Assume that...", "Let...", "Suppose..." or "Consider...".

At first, we get the formula image PremisesA of the antecedent PRS PrsA. Then we check PrsB with the concatenation of PremisesBegin and PremisesA if CheckTrigger is set to "check"; otherwise we just get its formula image PremisesB as well. Now we distinguish between two cases concerning the update of PremisesBegin:

The first case is that we deal with a proof by contradiction indicated by

```
type~relation..arity~0..name~'$false'
```

as the last premise of PrsB. In this case, if the antecedent PRS contains premises, we negate them - since a contradiction could be derived from the assumptions - and append the negated formulas to PremisesBegin, instantiating NewPremisesBegin with the result. If, however, PremisesA is empty, this means that the contradicting assumptions must have been made on a higher level. This occurs for example when the user writes "Let x be given." in a proof by contradiction. Therefore we instantiate NewPremisesBegin with the concatenation of PremisesBegin and the "false"-premise from above.

In the case that we deal with a "normal" assumption (i. e. no contradiction), the procedure is different, but depends on the antecedent PRS as well. If it contains no premises, we interpret the condition as a "for all..." statement; therefore, for every formula F in PremisesB, we skolemize the free variables in F using the predicate `skolemize` (see section 8) and append

$\forall VariablesA : F$

to PremisesBegin, where VariablesA are the variables in the *mrefs* of PrsA. If, however, the *mrefs* of PrsA are empty as well, something must have gone wrong and we throw the error message "Assumption PRS: Antecedent PRS A is empty and contains no variables". I have programmed the predicate `error_if_empty` that performs this task and is also used by other `check_condition` cases.

In the case that PremisesA is not empty, we skolemize the free variables of every formula F in PremisesB as we did in the first case and then append

$$\forall VariablesA : ConjunctPremisesA \Rightarrow F$$

to PremisesBegin, where VariablesA are again the variables in the *mrefs* of A and ConjunctPremisesA is the conjunction PremisesA.

The recursive call of `check_conditions` is the same as in the exclusive disjunction case.

8 The Module "premises.pl"

Most of the auxiliary predicates I wrote for *checker.pl* can be found in the file *premises.pl*.

For example, the predicate `quantify_existentially` conjuncts a list of formulae and quantifies over the variables in a list of mathematical referents (which are extracted by the predicate `variables`); `quantify_existentially` is used among others in the exclusive disjunction case of `check_conditions`. The predicate `make_induction_formulas` produces the formulae for the induction case of `check_prs` as described in section 7.3; similarly, `make_implication_formula`, `make_equivalence_formula` etc. are auxiliary predicates for the corresponding cases in `check_conditions`.

Furthermore, the predicates `update_skolem_variables` and `skolemize` can be found in *premises.pl*. I programmed them for the skolemization in the assumption case of `check_conditions` (see section 7.4). The procedure is the following:

In order to create and assign the skolemized variables in the first place, we use the predicate `update_skolem_variables`. Then the predicate `skolemize` is used in order to exchange the variables with their skolemized representatives.

`skolemize` has four arguments:

+ Formula:DOBSOD
+ Variables:list(DOBSOD)
+ SkolemVariables:list(DOBSOD)
- SkolemizedFormula:DOBSOD

and replaces each variable of Variables in the formula Formula by the corresponding skolemized variable which can be found in SkolemizedVariables, using my predicate `replace_in_formula`. An example for a skolemized variable is

```
type~function..name~skolem..arity~2
              ..args~[1,type~variable..name~x
                        ..arity~0..args~[]]
```

which means that this variable is the first one that depends on the (bound) variable x.

9 The Module "prs_export.pl"

The file *prs_export.pl* is responsible for the output of PRSs. At the moment, the output is given in HTML format but we plan to use XML format as well. There are two main reasons: firstly, it is more flexible; secondly, XML allows to represent a PRS in a universally processable format. Therefore I updated the predicate `prs_to_xml` which converts between the DOBSOD- and XML-Representation of a PRS. The XML representation of the PRS condition "x=y" can be found in the appendix.

10 Miscellaneous Work

I created the favicon (the little icon which can be seen next to the web address in the browser) for the Naproche website. At the moment, it appears only on the local Naproche website but we plan to use it in the online version later.

11 Conclusion

At the beginning, especially the inderdisciplinary nature of the Naproche Project appealed to me.

My expectations were fulfilled: Not only did I study an interesting application of logic, but also the time on the Naproche Project gave me the opportunity to study different subjects apart from mathematics. On the one hand, I learnt Prolog; on the other hand I, worked with linguistic techniques like PRS.

Furthermore, programming the code was a great experience for me since it provided me with the opportunity to contribute to current research.

In conclusion, I can say that I found the internship very fruitful and interesting.

12 Appendix

12.1 XML

The PRS condition "x=y" looks like this in XML format:

```
<name>=</name>
        <type>relation</type>
        <arity>2</arity>
        <args>
          <arg>
            <name>x</name>
            <type>variable</type>
            <arity>0</arity>
            <args/>
          </arg>
```

```
    <arg>
      <name>y</name>
      <type>variable</type>
      <arity>0</arity>
      <args/>
    </arg>
  </args>
```

12.2 Code

Here are some examples for code I have written.

Skolemize and update_skolem_variables

```
skolemize(Formula,[],_SkolemVariables,Formula) :- !.

skolemize(Formula,Variables,SkolemVariables,SkolemizedFormula) :-
        Variables = [Variable|Rest],
        SkolemId = skolem_id(Variable,SkolemVariable),
        member(SkolemId,SkolemVariables),
        replace_in_formula(Variable,Formula,SkolemVariable,FormulaTmp),
        skolemize(FormulaTmp,Rest,SkolemVariables,SkolemizedFormula).


update_skolem_variables(SkolemVariables, _SkolemArity,
        _VariablesA, [], SkolemVariables) :-!.

update_skolem_variables(SkolemVariables, SkolemArity,
                VariablesA, AdditionalVariables,
                NewSkolemVariables) :-
        % case that Variable already appears in SkolemVariables
        % we add a new skolem_id
        AdditionalVariables = [Variable|Rest],

        % add a new skolem_id for Variable
        skolem_index(Index),
        SkolemFunction = type~function..name~skolem
                        ..arity~SkolemArity..args~[Index|VariablesA],
        SkolemId = [skolem_id(Variable,SkolemFunction)],
        new_skolem_index(Index),

        append(SkolemVariables,SkolemId,SkolemVariablesTmp),
        update_skolem_variables(SkolemVariablesTmp,SkolemArity,
                        VariablesA,Rest,NewSkolemVariables).
```

```
new_skolem_index(Index) :-
        retract(skolem_index(Index)),
        Index = name~Name,
        atom_number(Name,NameN),
        succ(NameN, NewNameN),
        atom_number(NewName,NewNameN),
        assertz(skolem_index(type~constant..arity~0..name~NewName)).
```

The induction case in check_prs

```
check_prs(PRS,PremisesBegin,PremisesEnd,CheckTrigger) :-
        % Case that PRS is an induction PRS
        PRS = id~Id..conds~Conds..accafter~Accafter..mrefs~Mrefs..rrefs~Rrefs,
        member('induction',Rrefs),

        % Mrefs must not contain any variables
        variables(Mrefs,[]),

        % get the premises of the PRS and throw error message if they are empty
        check_conditions(Id,Accafter,Conds,[],PremisesPRS,nocheck),
        error_if_empty(PremisesPRS,Id,'Induction PRS is empty'),

        % PremisesPRS has to be one universally quantified formula
        PremisesPRS = [UnivQuantifiedFormula],
        (
        (UnivQuantifiedFormula = name~'!',!)
        ;
        (
        term_to_atom(Id,AtomId),
        add_error_message(logic,'check_prs',AtomId,
                'Wrong formula for induction PRS
                (must be universally quantified).'),
        !,
        fail
        )),

        % Make Induction Formulas
        make_induction_formulas(PremisesPRS,Formula1,FormulaSucc),

        atom_concat(Id,'induc_start',IdStart),
        atom_concat(Id,'induc_step',IdEnd),

        % Check Formulas if CheckTrigger = check
        check_formula_if_trigger(PremisesBegin,Formula1,IdStart,CheckTrigger),
        check_formula_if_trigger(PremisesBegin,FormulaSucc,IdEnd,CheckTrigger),
```

```
        % update PremisesBegin
        append(PremisesBegin,PremisesPRS,PremisesEnd),
        !.
```

The assumption case in check_conditions

```
% Case that A is empty
check_conditions(Id,Accafter,[ X |Rest],
                PremisesBegin,PremisesEnd,CheckTrigger) :-
        X = (A==>B),

        % Get PremisesA
        check_prs(A,[],PremisesA,nocheck),
        PremisesA = [],
        !,

        % Check B just with PremisesBegin (because A is empty)
        % If CheckTrigger = nocheck then don't check B, else do check it

        check_prs(B,PremisesBegin,PremisesBeginAndB,CheckTrigger),
        !,

        A = mrefs~MrefsA..id~IdA,

        % Two cases:
        % 1 : The user made a prove by contradiction
        % 2 : We deal with a normal assumption
        % % The last entry in PremisesB determines that
        ( append(_,[type~relation..arity~0..name~'$false'],
                        PremisesBeginAndB) ->
                (
                % Contradiction case
                % Add 'false' to PremisesBegin
                append(PremisesBegin,
                        [type~relation..arity~0..name~'$false'],
                        NewPremisesBegin)
                )
                ;
                (
                % For all Formulas X in PremisesB add
                % ! [Variables in MrefsA] : X
                % to our Premises storage
                variables(MrefsA,VariablesA),
                error_if_empty(VariablesA,IdA,
                        'Assumption PRS:
```

```
                        Antecedent PRS A is empty and contains no variables'),
                append(PremisesBegin,PremisesB,PremisesBeginAndB),
                update_assumption(PremisesBegin,
                        NewPremisesBegin,MrefsA,PremisesA,PremisesB)
                )
        ),

        !,
        check_conditions(Id,Accafter,Rest,
                NewPremisesBegin,PremisesEnd,CheckTrigger).

% Case that A is not empty
check_conditions(Id,Accafter,[ X |Rest],
                PremisesBegin,PremisesEnd,CheckTrigger) :-
        X = (A==>B),
        !,

        % Check A.
        % We use nocheck as these are Assumptions and need not be checked.
        check_prs(A,[],PremisesA,nocheck),
        !,

        % Check B with all the Premises from A included
        % If CheckTrigger = nocheck then don't check B, else do check it
        append(PremisesBegin,PremisesA,PremisesBeginAndA),
        check_prs(B,PremisesBeginAndA,PremisesBeginAndAAndB,CheckTrigger),
        !,

        A = mrefs~MrefsA,

        % Two cases:
        % 1 : The user made a prove by contradiction
        % 2 : We deal with a normal assumption
        % % The last entry in PremisesB determines that
        ( append(_,[type~relation..arity~0..name~'$false'],
                        PremisesBeginAndAAndB) ->
                (
                % Contradiction case
                %
                % Negate Assumption, quantify universally
                % over variables in MrefsA and append
                % the formula to PremisesBegin
                negate_formulas(PremisesA,NegatedA),
                quantify_universally(MrefsA,NegatedA,QNegatedA),
                append(PremisesBegin,QNegatedA,NewPremisesBegin)
```

```
        )
        ;
        (
        % For all Formulas X in PremisesB add
        % a new over all free variables in A
        % quantified Statement of the form
        % ! [Variables in MrefsA] : Fol_A -> X
        % to our Premises storage
        append(PremisesBeginAndA,PremisesB,PremisesBeginAndAAndB),
        update_assumption(PremisesBegin,
                NewPremisesBegin,MrefsA,PremisesA,PremisesB)
        )
),

!,
check_conditions(Id,Accafter,
        Rest,NewPremisesBegin,PremisesEnd,CheckTrigger).
```

References

1. Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language*. 1999.