Laboration Report

Logic for Computer Scientists, DD1351
Laboration 2

2024-11-15

Group members:
Malte Berg, maltebe@kth.se
Alexander Timsäter, timsater@kth.se

1. Introduction

In this laboration, computational logic is explored by using SWI-Prolog to verify and test predetermined proofs in formal natural deduction. In order to achieve this an algorithm which is able to test the validity of a proof is designed and implemented in Prolog. The proofs input format is displayed below:

```
%List of premisses
[neg(neg(imp(p, neg(p))))].

%Goal
neg(p).

%Proof steps:
[
  [1, neg(neg(imp(p, neg(p)))), premise ],
  [2, imp(p, neg(p)), negnegel(1)],
  [
    [3, p, assumption ],
    [4, neg(p), impel(3,2) ],
    [5, cont, negel(3,4) ]
  ],
  [6, neg(p), negint(3,5)]
].
```

Prolog predicates in the report are referenced as `name/arity`, where `name` refers to the predicate name, and `arity` refers to the number of arguments the predicate takes.

1.1. Sub-Questions

The laboration contains multiple sub-questions which are all answered in this report:

1. Write two non trivial proof using natural deduction which requires the use of boxes to be solved, of which one is correct and the other is incorrect.
2. Design an algorithm which is able to test the validity of the proof, demonstrate this by running through the previously written proofs.
3. Write an SWI prolog programme which implements the algorithm, test the algorithm by designing edge-cases and testing a provided test suite.
4. Discuss if the implementation could be used to generate sequent proofs and how it functions in restricted cases.

2. Literature Study

To make sure the laboration could properly be performed, Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems.* 2nd ed. was read up to, and including, page 27. This literature study provided all of the information that was needed to correctly implement the rules in natural deduction in the programme.

The lecture notes on the topic of Prolog were revisited as well to provide knowledge of how to implement the solution. The official prolog documentation was used as a source to understand specific predicates.

The laborations instructions were studied in order understand the task and be pointed in the right direction.

3. Method

According to the lab instructions, the first task was to create two non-trivial proofs: one correct and one incorrect. In order to grasp the structure of the given proofs the text files containing valid and invalid proofs provided with the assignment were examined first. After the formatting was studied, the two proofs were constructed and can be found in appendix A.1 and A.2, the correct proof and incorrect proof, respectively.

After the proofs were constructed, the desired functionality of the programme was established using pen and paper. Through this the it was discovered that some natural deduction rules were easier to implement than others. When examining the structure of

the proofs in the text-files, another realization was made, that being that all sub-proofs, or "boxes", were structured as nested lists, i.e. lists in lists, meaning it could be "filtered out", or rather not handled, in the algorithm if the general predicates of rules only handled single lines, which are structured as `[LineNumber, Formula/Statement, Rule]` and not lists of lists. This meant that the handling of boxes could be handled by other specific predicates.

Thereafter a prolog file that were to contain the code for the implementation was created to begin experimenting. From the laboration instructions, the given predicate`verify/1` was modified and inserted. This code read correctly the premises, goals and proofs in separate from the provided text files. The `verify/1` predicate was to call on a predicate `valid_proof/4`, taking the premises, goal and proof as arguments. In the created implementation, the predicate `valid_proof/4` has a base, and recursive case.

In order to check if a rule can be applied to the current read line, `valid_proof/4` calls on different predicates that have multiple different cases. This was since having all the rules be their own predicate with different names, made the content under the cases of `valid_proof/4` cluttered. The final result was three predicates being called on in every step, as well as either the recursive call or a check if the last line is reached and the formula or statement is equal to the goal, the base case.

The initial test that was performed was testing a very simple sequent with a simple proof. This was the given example from laboration instructions page 6.(containing `p` as the only premise, `p` as the goal, and using `p` in the proof as the only line, referring to it being a premise.) This meant that the first rule, or rather check, to implement was wether the formula or statement in the line is part of the premises. This was easily done by implementing a predicate `isprem/2`, that does a member check between the statement and `Prems`. Testing the constructed rule gave a pass to the simple proof.

After this, the remaining rules that does not rely on boxes were implemented using cases of the predicate `rules/2`, matching the lists to make sure the right parts of the line were handled, and the right parts were ignored with an underscore.

Every rule not using boxes has in common that a check is done whether the line number of the current line being examined is larger than the line or lines mentioned in the use of the rule. This check is to make sure that the lines mentioned are present in the proof above the current line. Along with the line checks, there is also a call to the `member/2` predicate with a specific match needed to be found. For instance, the case for double negation elimination, to be able to add on line 9 of the proof `[B, `ϕ`, negnegel(A)]`, there must be a line `[A, neg(neg(`ϕ`), _)]`. The way that the double negation has appeared is not relevant to the member check, thereby the underscore, as it is only of relevance whether there exists a line of the form wanted. The only exception to these common features is the LEM rule, which does not require anything previous in the proof to exist and can be added as long as it follows the form `or(`ϕ`, neg(`ϕ`))`.

After testing was performed to ensure that the rules worked for proofs without boxes, plans for how to check proofs relying on boxes were discussed. The first step was to make sure that rules relying on boxes had the correct start and end lines in the box that should be true for the different rules. Finally, way to check if the lines between the start and end lines were correct was to be implemented.

Starting with the way to help check whether boxes have the correct start and end lines, the predicate `last_line/2` was created. The predicate takes in two lists, the first being a list of lists, or in this case a box, and the second being a list in the form of the line wanted to be the end line. The predicate recursively continues through the first list until it is at the last line, or if the list is only one line, instantly, comes to the base case where the actual line is compared to the wanted last line. If this is true, then the call to the predicate comes out to true.

This predicate only checks the last line however, and in the cases of `rules/2` that rely on boxes, at least one call to `member/2` is needed as well. For instance, the PBC rule must start with `neg(`ϕ`)` and end with $\bot$. Therefor, the case of the predicate does a member check if there is a list of lists that starts with the specific line of a negated formula on the line `BoxStart`, referenced by the rule `pbc(BoxStart, BoxEnd)`, and if the predicate `last_line/2` comes out to true with the full box as its argument compared to a $\bot$ being found on line `BoxEnd`. If all of these steps come out to true, the PBC can be considered true. It is however clear that this only checks the first and last line of a box, and that the middle part must be independently checked some other way.

The final step was to verify the validity of boxes in the proof. As the predicate `rules/2` was designed to only handle non-nested lists, it will fail if given a nested list. A new predicate `valid_box/3` was created and added to the predicates called from `valid_proof/4` to handle these lists of lists. If the line handled is not a premise, and does not fit into any rule, the only possible way it is part of a valid proof is if it is a box, i.e. a list of lists.

The first thing the predicate does, regardless if it is only one list, or rather line, or many lists in the box, it checks if the first or only line is an assumption, as that is the only valid way of opening a box in formal natural deduction, and the proof naturally fails otherwise.

After the programme makes sure the first line is correct, if the box is longer than a single line, it appends all the lines in the box to the full proof and saves it to `SubProof`, which is used in a new call to `valid_proof/4` with an omitted goal, as it is only of interest whether the lines in the box are correctly used. Going through all of the steps in `valid_proof/4`, it is of course then possible that a call to the box handling predicate is done again, which then handles boxes in boxes effectively. If all of the predicate passes, it moves on to the next part of the proof after the box.

Using the given programme `run-all-tests` all of the given proofs were tested, passing all of the valid proofs and correctly identifying the invalid proofs provided as invalid. Along with this the author-constructed proofs containing boxes were both correctly handled, giving a pass to the correct proof and failing the incorrect proof.

4. Result

The algorithm was tested and proved to be effective through a manual paper run through. The efficacy of the algorithm was further shown in the programme, which was extensively tested using the provided test suite, as well as created proofs that were not provided.

5. Discussion

The programme was tested extensively using simpler proofs however more extreme proofs with hundreds, thousands or hundred of thousands of proof steps or nested boxes were not explicitly tested. In theory the designed code should work without error but has not been proven. The time complexity of the algorithm was not tested or considered during the design, this works with smaller proofs but if incredibly long proofs were to be verified issues might occur.

The programme manages to not only pass all of the given tests as well as author-constructed tests, but also to do it in a short amount of time.

This programme could be modified to instead of only checking proofs, also generate them, which can be visualised using a tree. The node would be a set of premises, and each node would be a set of antecedents. All of the valid possible rules that are applicable to that set are its edges. By traversing down this tree using a width-first approach while checking for the occurrence of the goal it can be used to generate proofs.

This programme already has the core logic for the root as this is simply the `isprem` predicate, if these were appended into a list:

```
%Premise!
isprem([_, A, premise], Prems):-
    member(A, Prems).
```

Parts of the needed node logic is already in use throughout the programme with the Linenum checks found in most rules. However all of the possible antecedents needs to be held track of by the programme, perhaps using a list.
Finally all of the edges from the root and all nodes can be calculated in a similar manner as to how the programme checks which rule is used in a line:

```
valid_proof(Prems, Goal, [X|T], FullProof):-
    (isprem(X, Prems); rules(X, FullProof);
```

```
    valid_box(Prems, X, FullProof)),
    valid_proof(Prems, Goal, T, FullProof).
```

To calculate all of the edges from the root or node it uses the set of all its antecedents and tries all of the possible rules for them, in which each possible rule is an edge.

This programmes core structure and approach would be able to be modified to generate proofs, however significant changes would be needed as mentioned above. Furthermore multiple "dead branch" checks, and loop-prevention mechanisms should be implemented in order to not waste computational resources.

Overall this would be an computationally intensive process as the number of edges and nodes at least exponentially increases leading to a high time complexity. For larger or more complicated proofs this approach would be computationally unfeasible without more focused methods.

An alternative method would be to go backwards from the goal and attempt to construct proof steps which generate the specific list of premises. This would follow a similar structure but potentially yields fewer steps, the rules predicates could be used without less changes. Finding new edges would work nearly identically as described above, however the initial list of antecedents would only contain the goal.

The main issue with this approach is that determining when the correct proof has been found, since it needs to check all of the proof steps for the specific premises. In an actual attempt this second approach would probably easier to implement using the written code due to the fewer number of changes to the overall logic that are needed.

A. Appendix

A.1. Correct Proof

```
[imp(neg(s), k), imp(k, n), imp(and(k, n), neg(s))].

or(and(k, neg(s)), and(neg(k), s)).

[
    [1, imp(neg(s), k), premise],
    [2, imp(k, n), premise],
    [3, imp(and(k, n), neg(s)), premise],
    [4, or(s, neg(s)), lem],
    [
        [5, s, assumption],
        [6, neg(neg(s)), negnegint(5)],
        [7, neg(and(k, n)), mt(3, 6)],
        [
            [8, k, assumption],
            [9, n, impel(8, 2)],
            [10, and(k, n), andint(8, 9)],
            [11, cont, negel(10, 7)]
        ],
        [12, neg(k), negint(8, 11)],
        [13, and(neg(k), s), andint(12, 5)],
        [14, or(and(k, neg(s)), and(neg(k), s)), orint2(13)]
    ],
    [
        [15, neg(s), assumption],
        [16, k, impel(15, 1)],
        [17, and(k, neg(s)), andint(16, 15)],
        [18, or(and(k, neg(s)), and(neg(k), s)), orint1(17)]
    ],
    [19, or(and(k, neg(s)), and(neg(k), s)), orel(4, 5, 14, 15, 18)]
].
```

A.2. Incorrect Proof

```
[imp(k,n)].

or(and(k,n), s).

[
    [1, imp(k,n), premise],
    [
        [2, k, assumption],
        [3, n, impel(1,2)],
        [4, and(k,n), andint(2,3)]
    ],
    [5, or(and(k,n), s), orint1(4)]
].

/*
    Line 5 is invalid, as it tries to access a
    line in a closed box to get to the goal.
*/
```

A.3. Full Code

```
verify(InputFileName):- %Reads a specified file
    see(InputFileName),
    read(Prems), read(Goal), read(Proof),
    seen,
    (valid_proof(Prems, Goal, Proof, Proof) ->
        write(' yes')
    ;
        write(' no'),
        !,
        fail
    ).

% Base case: Last line of proof
valid_proof(Prems, Goal, [X], FullProof):-
    (isprem(X, Prems); rules(X, FullProof);
    valid_box(Prems, X, FullProof)),
    X = [_,Goal,_]. % Check if last line of proof is Goal
% Recursive case: Not last line of proof
valid_proof(Prems, Goal, [X|T], FullProof):-
    (isprem(X, Prems); rules(X, FullProof);
    valid_box(Prems, X, FullProof)),
    valid_proof(Prems, Goal, T, FullProof).

% ======================================================================
% ======================Logical Rules, No Boxes=======================
% ======================================================================

%Implication Elimination!
rules([LineNum, B, impel(Line1, Line2)], FullProof):-
    LineNum > Line1, LineNum > Line2,
    member([Line1, A , _], FullProof),
    member([Line2, imp(A,B),_], FullProof).

%Negation Elimination!
rules([LineNum, cont, negel(Line1, Line2)], FullProof):-
    LineNum > Line1, LineNum > Line2,
    member([Line1, A, _], FullProof),
    member([Line2, neg(A), _], FullProof).

%And Introduction!
rules([LineNum, and(A,B), andint(Line1, Line2)], FullProof):-
    LineNum > Line1, LineNum > Line2,
```

```
    member([Line1, A, _], FullProof),
    member([Line2, B, _], FullProof).

%Double Negation Elimination!
rules([LineNum, A, negnegel(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, neg(neg(A)), _], FullProof).

%Double Negation Introduction!
rules([LineNum, neg(neg(A)), negnegint(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, A, _], FullProof).

%Modus Tollens!
rules([LineNum, neg(A), mt(Line1, Line2)], FullProof):-
    LineNum > Line1, LineNum > Line2,
    member([Line1, imp(A, B), _], FullProof),
    member([Line2, neg(B), _], FullProof).

%LEM!
rules([_, or(A, neg(A)), lem], _).

%Contradiction Elimination!
rules([LineNum, _, contel(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, cont, _], FullProof).

%Copy!
rules([LineNum, A, copy(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, A, _], FullProof).

%And Elimination Left!
rules([LineNum, A, andel1(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, and(A,_), _], FullProof).

%And Elimination Right!
rules([LineNum, A, andel2(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, and(_,A), _], FullProof).

%Or Introduction Left!
rules([LineNum, or(A, _), orint1(LineA)], FullProof):-
```

```
    LineNum > LineA,
    member([LineA, A, _], FullProof).

%Or Introduction Right!
rules([LineNum, or(_, A), orint2(LineA)], FullProof):-
    LineNum > LineA,
    member([LineA, A, _], FullProof).

% ====================================================================

% ====================================================================
% ======================Logical Rules, With Boxes!====================
% ====================================================================

%Implication Introduction!
rules([LineNum, imp(A,B), impint(BoxStart,BoxEnd)], FullProof):-
    LineNum > BoxStart, LineNum > BoxEnd,
    member([[BoxStart, A, assumption]|Box], FullProof),
    last_line([_|Box], [BoxEnd, B, _]).

%Negation Introduction!
rules([LineNum, neg(A), negint(BoxStart,BoxEnd)], FullProof):-
    LineNum > BoxStart, LineNum > BoxEnd,
    member([[BoxStart, A, assumption]|Box], FullProof),
    last_line([_|Box], [BoxEnd, cont, _]).

%PBC!
rules([LineNum, A, pbc(BoxStart,BoxEnd)], FullProof):-
    LineNum > BoxStart, LineNum > BoxEnd,
    member([[BoxStart, neg(A), assumption]|Box], FullProof),
    last_line([_|Box], [BoxEnd, cont, _]).

%Or Elimination!
rules([LineNum, X, orel(OrLine,Box1Start,Box1End,Box2Start,Box2End)], FullProof):-
    LineNum > OrLine, LineNum > Box1End, LineNum > Box2End,
    member([OrLine, or(A,B), _], FullProof),
    member([[Box1Start, A, assumption]|Box1], FullProof),
    last_line([_|Box1], [Box1End, X, _]),
    member([[Box2Start, B, assumption]|Box2], FullProof),
    last_line([_|Box2], [Box2End, X, _]).

% ====================================================================

% ====================================================================
```

```
% ========================Helper Predicates=========================
% ==================================================================

%Premise!
isprem([_, A, premise], Prems):-
    member(A, Prems).

% Check if valid box! <3
%Base Case!
valid_box(_, [Line], _):-
    Line = [_, _, assumption].
%Recursive Case!
valid_box(Prems, [Line|Rest], FullProof):-
    Line = [_, _, assumption],
    append([Line|Rest], FullProof, SubProof),
    valid_proof(Prems, _, Rest, SubProof).

%Used for finding last line!
%Base Case!
last_line([Line], Line).
%Recursive Case!
last_line([_|Rest], LastLine):-
    last_line(Rest, LastLine).

% ==================================================================
```

A.4. Table of Helper Predicates

Predicate	**True When**	**False When**
verify	File is correctly formatted eg. contains a list, a variable, and a list in that order. `valid_proof` suceeds.	File can not be read or `valid_proof` fails
valid_proof (base)	Last line is either premise or valid rule, and contains Goal from verify.	Line is not premise or valid rule, or doesn't contain Goal
valid_proof (recursive)	Current line of proof is valid.	Current line of proof is invalid.
isprem	Formula is found in the list of premises.	Formula is not found in the list of premises
valid_box (base)	Single line with assumption.	Line isn't an assumption
valid_box (recursive)	First line is assumption and rest is valid proof.	First line not assumption or rest invalid
last_line (base)	Line matches the wanted line provided.	List empty or line does not match the wanted line.
last_line (recursive)	Box has one or more remaining lines	List empty

Table 1: Truth Conditions for Helper Predicates

A.5. Table of Rules Cases

"Current line number is larger than line number(s) referenced by used rule" = A
"The list used as the first argument must match the correct formula/statement and rule used in a line, and only reference lines available outside of closed boxes." = B

These rules are used in the "True When" part of Table 2.

Case Rule	**True When**	**False When**
impel(P, Q)	A, B. Order of the lines referenced in the rule are ϕ at line P, and `imp(`ϕ`,` ψ`)` on line Q.	Order is switched in line references.
negel(P, Q)	A, B. Line P must contain ϕ and Line Q must contain neg(ϕ).	Order is switched in line references.
andint(P, Q)	A, B. Line P must contain ϕ and line Q must contain ψ to get formula `and(`ϕ`,` ψ`)`.	Order is switched in line references.
negnegel(P)	A, B. `neg(neg(`ϕ`))` must be found on line P to get formula ϕ.	Formula not found on line P.
negnegint(P)	A, B. ϕ must be found on line P to get formula `neg(neg(`ϕ`))`.	Formula not found on line P.
mt(P, Q)	A, B. Line P contains `imp(`ϕ`,` ψ`)` and line Q contains `neg(`ψ`)` to get `neg(`ϕ`)`.	Order is switched in line references, formulas not found on referenced lines.
lem	Always true for `or(`ϕ`,neg(`ϕ`))`	Order on formulas are switched.
contel(P)	A, B. $\perp$ exists on line P.	$\perp$ not found on line P.
copy(P)	A, B. Line P contains ϕ.	ϕ is not found on line P.
andel1(P)	A, B. Line P contains and(ϕ, _).	and(ϕ, _) is not found, the order of ϕ and _ is swapped. All other cases.
andel2(P)	A, B. Line P contains and(_, ψ).	and(_, ψ) is not found, the order of ψ and _ is swapped.
orint1(P)	A, B. Line P contains ϕ, and `or(`ϕ`,` ψ`)` is found on current line.	ϕ is not found on line P.
orint2(P)	A, B. Line P contains ϕ, and `or(`ψ`,` ϕ`)` is found on current line.	ϕ is not found on line P.
impint(P, Q)	A. Box starts with ϕ as assumption on line P and ends with ψ on line Q to get `imp(`ϕ`,`ψ`)`.	Order is switched in line references, box starts or ends incorrectly.
negint(P, Q)	A. Box starts with ϕ as assumption on line P and ends with $\perp$ on line Q to get `neg(`ϕ`)`.	Order is switched in line references, box starts or ends incorrectly.
pbc(P, Q)	A. Box starts with `neg(`ϕ`)` as assumption on line P and ends with $\perp$ on line Q to get ϕ.	Order is switched in line references, box starts or ends incorrectly.
orel(P, Q, R, S, T)	A. Line P contains `or(`ϕ`,`ψ`)`, line Q contains ϕ line R contains X, line S starts with ψ and ends with X	Order is switched in line references, boxes starts or ends incorrectly, formula not found on line P

Table 2: Truth Conditions for Logical Rules