Prolog Proof Control

Logik för dataloger, DD1351

Ammar Alzeno

29/11 - 2023

Introduction

The primary objective of this project was to develop a Prolog algorithm capable of checking the correctness of proofs constructed using natural deduction rules. This entails the algorithm being able to parse and interpret a structured representation of proofs, apply natural deduction rules, and determine the validity of each step in the proof sequence. Prolog's inherent structure, grounded in formal logic, makes it very suitable for tasks involving logical inference and validation.

The approach will be as following, predicates will be introduced to traverse each proof, validating line by line, that the used rules of natural deduction are being used correctly. When a line is correctly used, it will be added to a list of validated lines so that future callbacks only take in validated information. When all lines have been validated, the last line will be cross-checked with the goal, if that is true then the proof is valid. The method below will show indepth implementation of this approach.

Method

The Prolog algorithm that will be developed for verifying natural deduction proofs will require a comprehensive system designed to parse, interpret, and validate structured proofs. It will also require validation of boxes (subproofs) as well as nested boxes. We will go over each part of the algorithm below, describing the methodology behind every aspect of the algorithm.

Input Processing: The algorithm will begin with verify(InputFileName), which will serve as the entry point. This predicate is responsible for reading the proof data from a specified file. The file contains premises (Prems), a goal (Goal), and the proof itself (Proof). The see and seen predicates manage file reading in Prolog, ensuring that the proof data is correctly imported for further processing. This code was provided in the instructions.

Valid Proof Verification

Base Case Handling: The valid_proof predicate is pivotal in the algorithm. The base case for this recursive predicate will be defined to handle an empty proof, if the proof is not empty it will validate the proof through valid_proof_lines. It will then call valid_goal, which is described below. This step is crucial for confirming that the proof conclusively achieves its intended objective.

Goal Validation: The valid-goal predicate will check that the goal of the proof corresponds to the conclusion of the last line, and that this last line is not merely an assumption. This is essential to ensure that the proof logically arrives at the desired conclusion.

```
% Main function to validate the proof
valid_proof(Prems, Goal, Proof) :-
    valid_goal(Goal, Proof), % Check if the goal is achieved in the last line
    valid_proof_lines(Prems, Proof, []). % Validate each line of the proof
```

Line-by-Line Proof Validation

Regular Line Handling: When dealing with non-box proof lines, valid_proof_lines will invoke valid_line for each line. This predicate is responsible for applying the appropriate natural deduction rules to validate each line in the context of the proof.

Empty Proof Case: For an empty proof, valid_proof_lines will ensure that no further validation is needed, signifying the end of the proof.

Handling Boxes: The algorithm will extend the valid-proof-lines predicate to manage boxes, which are essential for nested subproofs and assumptions. Each box is represented as a list within the proof, starting with an assumption. The predicate first validates the lines within the box (BoxTail) recursively, treating the assumption as the initial line of the validated lines for the box. After validating the box, it continues to validate the rest of the proof outside the box (ProofTail), including the entire box as part of the validated lines. Each rule uses boxes differently, more on this later.

Rule Validation

Rule validation will be taken care of by valid-line, which is recursively called in the earlier steps. There will be a claus for each natural deduction rule. Different rule implementations were needed in order to handle boxes in the algorithm. An example below of the rule implementation for 'copy(x)' shows a more simple implementation.

```
% Implementation for the 'copy(x)' rule
valid_line(_Prems, [_Row, Statement, copy(X)], ValidatedLines) :-
    member([X, Statement, _], ValidatedLines).
% See Appendix B for the rest of the rules
```

Box Identification and Processing

Box as a List: In the algorithm, a box will represented as a list of proof lines, distinguishable from regular lines. This distinction is crucial for processing nested logical structures.

find_line Predicate: This predicate will be a key component for navigating through the proof, including within boxes. It recursively searches for a specific line number (Nr) throughout the entire proof structure, including inside any boxes. This allows the algorithm to reference lines both within and outside of subproofs when applying certain deduction rules.

find_line_in_box Predicate: This predicate is used for finding lines within a specific box. It's crucial for rules that require both the starting and ending lines of a subproof, ensuring that the referenced lines indeed belong to the same logical unit.

```
find_line_in_box(Start, End, ValidatedLines, [StartStatement, EndStatement]) :-
    member(Box, ValidatedLines),
    is_list(Box),
    member([Start, StartStatement, _], Box),
    member([End, EndStatement, _], Box).
```

Application of Rules within Boxes

Handling Assumptions: Assumptions are the starting points of boxes. The algorithm ensures that any line marked as an assumption begins a new box.

Rule Validation with Box Context: Many deduction rules, like andint, impint, negint, orel, and others, will be adapted to consider the context of boxes. For example, the andint rule checks if both components are found either outside any box or within the same box. This adaptation is critical for maintaining logical consistency, particularly in proofs involving nested assumptions. However not all rules need to be specifically adapted.

Specific Rule Implementations

```
% Implementation for the 'pbc(x,y)' rule, accounting for boxes
valid_line(_Prems, [_Row, Conclusion, pbc(X, Y)], ValidatedLines) :-
    find_line_in_box(X, Y, ValidatedLines, [neg(Conclusion), cont]).

% Implementation for the 'negint(x,y)' rule, accounting for boxes
valid_line(_Prems, [_Row, neg(Assumption), negint(X, Y)], ValidatedLines) :-
    find_line_in_box(X, Y, ValidatedLines, [Assumption, cont]).
```

'negint' and 'pbc' Rules: The two examples above use find_line_in_box to account for boxes. These rules heavily rely on box context. negint looks for a contradiction (cont) within the same box as the assumption. Similarly, pbc (proof by contradiction) searches for a contradiction within a box that starts with the negation of the conclusion. For the rest of the rules and their box implementations, see Appendix B.

Result

After implementing all the different predicates explained in the method, the algorithm successfully passed all the tests provided, as well as two extra tests seen below.

```
% Example of a non-trivial proof that should return true
% Premises
[ imp(p, q), imp(q, r), imp(r, s), neg(s) ].
% Goal
neg(p).
% Proof
[    [1, imp(p, q), premise],
    [2, imp(q, r), premise],
    [3, imp(r, s), premise],
    [4, neg(s), premise],
    [
        [5, p, assumption],
        [6, q, impel(5, 1)],
        [7, r, impel(6, 2)],
        [8, s, impel(7, 3)],
        [9, cont, negel(8, 4)]
    ],
    [10, neg(p), negint(5, 9)]
].
```

After plugging in this input file, we get a 'true'. That is indeed correct, here is a breakdown of the correct usage of the rules: impel (Implication Elimination) is applied three times to derive q from p, r from q, and s from r. negel (Negation Elimination) is used to derive a contradiction from s and neg(s). negint (Negation Introduction) is then used to introduce neg(p) based on the contradiction derived in the subproof.

```
% Example of a non-trivial proof that should return false
% Premises
[ imp(p, q), imp(q, r), r ].
% Goal
or(neg(p), s).
% Proof
```

```
[
    [1, imp(p, q), premise],
    [2, imp(q, r), premise],
    [3, r, premise],
    [
        [4, p, assumption],
        [5, q, impel(4, 1)],
        [6, r, impel(5, 2)],
        [7, s, copy(6)]
    ]
    [8, imp(p, s), impint(4, 7)],
    [9, or(neg(p), s), orint2(8)]
].
```

A quick breakdown of the input: copy incorrectly assumes s from r, which is logically invalid, impint (Implication Introduction) also incorrectly introduces imp(p, s) based on the invalid assumption. Lastly orint2 (Or Introduction 2) incorrectly introduces or(neg(p), s) using an invalid implication. The misuse of these rules and incorrect assumptions lead to the algorithm outputting a 'false'. Below is the results of all the tests in Figure 1.

valid01.txt	OK	valid23.txt	OK	invalid20.txt	OK
valid02.txt	OK	invalid01.txt	OK	invalid21.txt	OK
valid03.txt	OK	invalid02.txt	OK	invalid22.txt	OK
valid04.txt	OK	invalid03.txt	OK	invalid23.txt	OK
valid05.txt	OK	invalid04.txt	OK	invalid24.txt	OK
valid06.txt	OK	invalid05.txt	OK	invalid25.txt	OK
valid07.txt	OK	invalid06.txt	OK	invalid26.txt	OK
valid08.txt	OK	invalid07.txt	OK	invalid27.txt	OK
valid09.txt	OK	invalid08.txt	OK	invalid28.txt	OK
valid10.txt	OK	invalid09.txt	OK	invalid29.txt	OK
valid11.txt	OK	invalid10.txt	OK	invalid30.txt	OK
valid12.txt	OK	invalid11.txt	OK	invalid31.txt	OK
valid13.txt	OK	invalid12.txt	OK	invalid32.txt	OK
valid14.txt	OK	invalid13.txt	OK	invalid33.txt	OK
valid15.txt	OK	invalid14.txt	OK	invalid34.txt	OK
valid16.txt	OK	invalid15.txt	OK	invalid35.txt	OK
valid17.txt	OK	invalid16.txt	OK	invalid36.txt	OK
valid18.txt	OK	invalid17.txt	OK		
valid19.txt	OK	invalid16.txt	OK		
valid20.txt	OK	invalid17.txt	OK		
valid21.txt	OK	invalid18.txt	OK		
valid22.txt	OK	invalid19.txt	OK		

Figure 1: Results

Appendix A

All the predicates in the algorithm, as well as explanations for when they are true/false.

Predicate	True	False
verify/1	• The file specified by **InputFileName** exists and can be opened for reading. • The premises, goal, and proof can be successfully read from the file. • The proof is valid according to the natural deduction rules, as determined by the **valid_proof/3** predicate.	Otherwise false
valid_proof/3	• **Base Case**: If the proof is empty and the goal has been achieved in the last line validated prior to calling this base case. • **General Case**: If the goal is achieved in the last line of the proof **(valid_goal/2)** and every line in the proof is valid according to the rules of natural deduction **(valid_proof_lines/3)** .	Otherwise false
valid_goal/2	• The last line of the proof achieves the goal (**Goal** matches the second element of the last proof line). • The rule used in the last line is not an assumption **(Rule \= assumption).**	Otherwise false

Figure 2: Predicates

valid_proof_lines/3	• **Base Case**: True when there are no more lines to validate (empty proof). • **Regular Lines**: True when each line in the proof, including the current line and all following lines, is valid according to **valid_line/3.** • **Boxes**: True when all lines within each box and the remaining lines of the proof outside the boxes are valid.	Otherwise false
valid_line/3	• If a given line is the result of the (correct) use of one of the 14 rules on another validated line. **OR** • If the line contains a premise that is in the premises list.	Otherwise false
find_line/3	• The predicate is true when a line with the number Nr is found within the **ProofLines.** It can be a regular line or a line within a box.	Otherwise false
find_line_in_box/4	• The predicate is true if both the **Start** and **End** lines are found within the same box in **ValidatedLines.** • It successfully retrieves the corresponding statements for these lines.	

Figure 3: Predicates (cont.)

Appendix B

The full code for the algorithm.

```
verify(InputFileName) :-
  see(InputFileName),
  read(Prems), read(Goal), read(Proof),
  seen,
  valid_proof(Prems, Goal, Proof).


%%%%%%%%%%%%%%%% goal %%%%%%%%%%%%%%

% Check that the goal of the proof is the conclusion of the last line in
↪  the proof
% and that the last line is not an assumption
valid_goal(Goal, Proof) :-
    last(Proof, [_Row, Goal, Rule]),
    Rule \= assumption.


%%%%%%%%%%%%%%%% valid_proof %%%%%%%%%%%%%%%%

% Base case for valid_proof: a proof is valid if it is empty and the last
↪  validated line achieves the goal
valid_proof(_Prems, Goal, [], [[_Row, Goal, _Rule]|_Validated]).


% Main function to validate the proof
valid_proof(Prems, Goal, Proof) :-
    valid_goal(Goal, Proof), % Check if the goal is achieved in the last
    ↪  line
    valid_proof_lines(Prems, Proof, []). % Validate each line of the
    ↪  proof


%%%%%%%%%%%%%%%% valid_proof_lines %%%%%%%%%%%%%%%%

% The base case for an empty proof (no more lines to validate)
valid_proof_lines(_Prems, [], _ValidatedLines).

% Handling regular proof lines (non-box)
valid_proof_lines(Prems, [ProofLine | RestOfProof], ValidatedLines) :-
    valid_line(Prems, ProofLine, ValidatedLines),
    valid_proof_lines(Prems, RestOfProof, [ProofLine | ValidatedLines]).

% Extend valid_proof_lines to handle boxes
```

```
valid_proof_lines(Prems, [[[Row, Assumption, assumption] | BoxTail] |
↪  ProofTail], ValidatedLines) :-
    valid_proof_lines(Prems, BoxTail, [[Row, Assumption, assumption] |
    ↪  ValidatedLines]),
    valid_proof_lines(Prems, ProofTail, [[[Row, Assumption, assumption] |
    ↪  BoxTail] | ValidatedLines]).


%%%%%%%%%%%%%% find_line %%%%%%%%%%%%%%%

% Helper predicate to find a line in the validated lines, including
↪  within boxes
find_line(Nr, [Line|_], Line) :-
    Line = [Nr, _, _].
find_line(Nr, [Box|_], Line) :-
    is_list(Box),
    member(Line, Box),
    Line = [Nr, _, _].
find_line(Nr, [_|Rest], Line) :-
    find_line(Nr, Rest, Line).


%%%%%%%%%%%%%%%% find_line_in_box %%%%%%%%%%%%%%%%

% Helper predicate to find a line in a specific box
find_line_in_box(Start, End, ValidatedLines, [StartStatement,
↪  EndStatement]) :-
    member(Box, ValidatedLines),
    is_list(Box),
    member([Start, StartStatement, _], Box),
    member([End, EndStatement, _], Box).


%%%%%%%%%%%%%%%%% valid_line %%%%%%%%%%%%%%%%%


% Implementation for the 'premise' rule
valid_line(Prems, [_Row, Statement, premise], _ValidatedLines) :-
    member(Statement, Prems).

% Implementation for the 'copy(x)' rule
valid_line(_Prems, [_Row, Statement, copy(X)], ValidatedLines) :-
    member([X, Statement, _], ValidatedLines).
```

```
% Implementation for the 'andel1(x)' rule, accounting for boxes
valid_line(_Prems, [_Row, Statement1, andel1(X)], ValidatedLines) :-
    find_line(X, ValidatedLines, [X, and(Statement1, _), _]).


% Implementation for the 'andel2(x)' rule, accounting for boxes
valid_line(_Prems, [_Row, Statement2, andel2(X)], ValidatedLines) :-
    find_line(X, ValidatedLines, [X, and(_, Statement2), _]).

% Implementation for the 'orint1(x)' rule, accounting for boxes
valid_line(_Prems, [_Row, or(Statement, _), orint1(X)], ValidatedLines)
↪  :-
    find_line(X, ValidatedLines, [X, Statement, _]).


% Implementation for the 'orint2(x)' rule, accounting for boxes
valid_line(_Prems, [_Row, or(_, Statement), orint2(X)], ValidatedLines)
↪  :-
    find_line(X, ValidatedLines, [X, Statement, _]).

% Implementation for the 'orel(x,y,u,v,w)' rule, accounting for boxes
valid_line(_Prems, [_Row, Conclusion, orel(X, Y, U, V, W)],
↪  ValidatedLines) :-
    find_line(X, ValidatedLines, [X, or(Statement1, Statement2), _]),
    find_line_in_box(Y, U, ValidatedLines, [Statement1, Conclusion]),
    find_line_in_box(V, W, ValidatedLines, [Statement2, Conclusion]).


% Implementation for the 'impint(x,y)' rule, ensuring both x and y are
↪  within the same box
valid_line(_Prems, [_Row, imp(Assumption, Conclusion), impint(X, Y)],
↪  ValidatedLines) :-
    member(Box, ValidatedLines),
    is_list(Box),
    member([X, Assumption, assumption], Box),
    last(Box, [Y, Conclusion, _]).

% Implementation for the 'impel(x,y)' rule
% Ensuring x is a valid premise and y is an implication that implies the
↪  conclusion.
valid_line(_Prems, [_Row, Conclusion, impel(X, Y)], ValidatedLines) :-
    % Check if X is a valid premise or derived statement, not an
    ↪  assumption in a box
    member([X, Premise, _], ValidatedLines), \+ is_list(Premise),
```

```
    % Check if Y is an implication of the form imp(Premise, Conclusion)
    member([Y, imp(Premise, Conclusion), _], ValidatedLines).

% Implementation for the 'negint(x,y)' rule, accounting for boxes
valid_line(_Prems, [_Row, neg(Assumption), negint(X, Y)], ValidatedLines)
↪  :-
    find_line_in_box(X, Y, ValidatedLines, [Assumption, cont]).

% Implementation for the 'negel(x,y)' rule, accounting for boxes
valid_line(_Prems, [_Row, cont, negel(X, Y)], ValidatedLines) :-
    find_line(X, ValidatedLines, [X, Statement, _]),
    find_line(Y, ValidatedLines, [Y, neg(Statement), _]).

% Implementation for the 'contel(x)' rule, accounting for boxes
valid_line(_Prems, [_Row, _Conclusion, contel(X)], ValidatedLines) :-
    find_line(X, ValidatedLines, [X, cont, _]).

% Implementation for the 'negnegint(x)' rule
valid_line(_Prems, [_Row, neg(neg(Statement)), negnegint(X)],
↪  ValidatedLines) :-
    % Check that X refers to a proven statement and not an assumption in
    ↪  a box
    member([X, Statement, _], ValidatedLines), \+ is_list(Statement).

% Implementation for the 'negnegel(x)' rule
valid_line(_Prems, [_Row, Statement, negnegel(X)], ValidatedLines) :-
    % Check that X refers to a double negation of the statement, and is a
    ↪  proven statement
    member([X, neg(neg(Statement)), _], ValidatedLines), \+
    ↪  is_list(neg(neg(Statement))).

% Implementation for the 'mt(x,y)' rule, accounting for boxes
valid_line(_Prems, [_Row, neg(Antecedent), mt(X, Y)], ValidatedLines) :-
    find_line(X, ValidatedLines, [X, imp(Antecedent, Consequent), _]),
    find_line(Y, ValidatedLines, [Y, neg(Consequent), _]).

% Implementation for the 'pbc(x,y)' rule, accounting for boxes
valid_line(_Prems, [_Row, Conclusion, pbc(X, Y)], ValidatedLines) :-
    find_line_in_box(X, Y, ValidatedLines, [neg(Conclusion), cont]).

% Implementation for the 'lem' rule
valid_line(_Prems, [_Row, or(Statement, neg(Statement)), lem],
↪  _ValidatedLines).
```

```
% Implementation for 'andint'. Considers the scope of assumptions and
↪  boxes
valid_line(_Prems, [_Row, and(Statement1, Statement2), andint(X, Y)],
↪  ValidatedLines) :-
    % Find the lines corresponding to X and Y in the proof
    find_line(X, ValidatedLines, [X, Statement1, _]),
    find_line(Y, ValidatedLines, [Y, Statement2, _]),
    % Ensure both components are either before or after any assumption
    % Or one is an assumption and the other comes after
    (
        % Case 1: Both X and Y are before an assumption
        \+ (member(Box, ValidatedLines), is_list(Box), (member([X, _, _],
        ↪  Box); member([Y, _, _], Box)))
    ;
        % Case 2: Both X and Y are after an assumption and in the same
        ↪  box
        member(Box, ValidatedLines), is_list(Box),
        find_line_in_box(X, Y, ValidatedLines, [Statement1, Statement2])
    ;
        % Case 3: One is an assumption and the other comes after the
        ↪  assumption
        (member([X, _, assumption], ValidatedLines), Y > X) ;
        (member([Y, _, assumption], ValidatedLines), X > Y)
    ).
```