DD1350 Logic for Computer Science

Lab 1: Proof Control with Prolog

Authors: Arsalan Syed and Martin Nilsson.

Introduction

The purpose of this report is to create an algorithm using the programming language Prolog that can verify whether a proof in natural deduction is valid or not. This report explains how the algorithm works generally as well as the purpose of each predicate in the code.

Definitions

This sections will define certain terminology utilised within this lab report and the text.

Line: A line within the proof is represented as a list in Prolog containing a line number, a term and a function. It has the following syntax: [LineNum,Term,Func].

Line Number: Shows where in the proof the current line is located. It is written as LineNum in the code.

Term: Atoms representing a premise, assumption or result of rules applied on earlier lines in the proof (for example and(p,q), imp(neg(q)),q, etc).

Function: Represents how the term on this line was obtained. It can be a premise, assumption or rule (for example andint(1,2),lem). It is written as Func in the code.

Assumption box: A list where the first element in a line that contains an assumption. The last line is referred to as the conclusion of the box.

Assumption based rule: Any rule that requires an assumption and a conclusion. Only the rules orel,impint,negint and pbc have this quality.

Non Assumption based rules: All other rules that do not use assumptions.

General Description of the Algorithm

The algorithm starts by reading a text file which has three different segments, the premises, the goal and the proof. It will then check if these three segments form a valid proof (using the valid_proof predicate). It checks the last line in the proof to see whether or not it is the same as the goal that was specified. Then the algorithm iterates over each line in the proof using checkProof and checkProofLine. The predicate checkProof will check if a proof as a whole is true by calling checkProofLine on each line in the proof. The use of having checkProof is so that we can also treat assumption boxes as individual proofs. The predicate checkProofLine has several cases, depending on whether the current line is a premise, assumption or rule.

The different cases for checkProofLine

Using separate cases for checkProofLine allows us to use the same predicate and target it at different types of lines within the proof. When the current line is a premise, we search the list of premises to see if the term in the current line exists there. When we have to deal with an assumption, we call checkProof on that assumption box (and we skip the line where the assumption shows up).

In Prolog, items of a list can be predicates. For example, each line can have in most cases, have a rule in natural deductions such as impint(3,6) for example. Prolog has the capability of calling a predicate impint(X,Y) directly. The only problem is the this call only has information about the lines we are calling to but nothing about the line we are calling from or about the proof. To solve this, we can use the call predicate which can add extra arguments to a function call. This is implemented within checkProofLine so that we do not need to make a case for every rule.

There are specific cases for the rule lem and assumption based rules. Lem does not require any lines as arguments so it is called in a separate manner. Assumption based rules use a different predicate for argument verification which will be explained in the following sections.

Handling Assumption Boxes

When dealing with proofs, it is common to come across assumptions and it is vital that a verification program can handle assumptions correctly. Assumption boxes can be considered to be proofs of their own, they can use premises, contain assumptions or use rules. By treating the box as a proof, we have to check that every line from the assumption to the conclusion is valid. We also have restrictions when dealing with assumption boxes. All lines within a proof are not allowed to apply a rule with arguments lying inside an assumption box. This is because all formulas within an assumption box only exist within the scope of that box. All formulas deduced due to an assumption will only exist because of that assumption. If the validity of the assumption is uncertain, we cannot be sure that any formulas derived from it are also valid.

However, there is one situation where we can refer to something within an assumption box and that is when using assumption based rules. These rules check the assumption made and what conclusion it leads to. The reason why this is allowed is because these rules simply need to see if an assumption leads to a desired conclusion. So for rules such as orel and negint, we can refer to inside an assumption box but only to the first and last line in that box. Assumption based rules have another restriction which is that they do not call to an assumption box within an assumption box.

Predicates for Checking Argument Correctness

It is important that before any rules are called in a proof by natural deduction that we check if the arguments comply to conventions and use proper syntax.

First of all, a rule cannot call a line that occurs later in the sequence. Referring to a formula which has not even be deduced would be erroneous. For this case, the program uses a predicate, checkArgs which checks that the line we are calling from is greater than the lines in the arguments. For example, checkArgs(5,andint(2,3)) will pass since 5 is greater than 2 and 3 whereas checkArgs(2,copy(4)) will fail since 2 is not greater than 4. Also note that we cannot call to the same line in a proof that we are calling from. If andint(1,2) is called from line 2, this will fail.

Second of all, we have to deal with calls involving assumption boxes. Generally, we use the predicate checkLevels to see if a non assumption based rule is calling a line within an assumption box. We can introduce the concept of a level where anything that is not inside a box is at the base level of a proof. The base level is represented with 0 and for every assumption box we enter, we add 1 to the level.

```
[q].

imp(p,q).

[
  [1, q, premise],
  [
    [2, p, assumption],
    [
      [3, r, assumption],
      [4, q, copy(1)]
    ],
    [5, q, copy(1)]
  ],
  [6, imp(p,q), impint(2,5)]
].
```

Using the code to the left as an example, lines 1 and 6 in the proof are in level 0. Lines 2 and 5 are in level 1 and lines 3 and 4 are in level 2.

If N is the line we are calling from and X,Y are lines we are calling to, then checkLevels will be true if N is greater than or equal to X and Y individually. Having a greater level value means that you are calling to something outside of the box the call is coming from. Having an equal level value means that you are calling to something within the same box.

To handle the case for assumption based rules, we use a separate predicate, checkLevelsAssumption. It works similarly to checkLevels but instead it checks that the line we are calling from and the arguments have a level difference that is equal to one. In the figure above, line 6 is able to call to lines 2 and 5 but is prohibited from calling lines 3 or 4 since it has a level difference of two.

Verifying Rules

Each rule is generally verified in a similar manner. We use findInProof to find the terms in the lines that the arguments point to. From here, we use unification and look for specific patterns that each rule has to fulfil. For the case of andint(2,3) being called from line 4, we have to see that whatever is at line 4 is of the form and(p,q) where p and q lie on lines 2 and 3 respectively. We also used predicates that will always fail in the case that something that is not a rule gets treated as a rule. If a line has assumption as their function but the line does not lie within a box, then assumption will be treated as a rule. When this case occurs, the predicate assumption(X,Y) will fail automatically.

Tables for Explaining Predicate Purposes

The following tables explain what every defined predicates in the proof does and when it is true and when it is false.

Table 1: This table covers the main predicates related to iterating over the proof and checking each line in the proof.

Predicate	**When Predicate is True**	**When Predicate is False**	**The use of this predicate.**
verify	When a given text file contains a valid proof	When a given text file contains an invalid proof	Can check a test file to see if it has a valid proof
valid_proof	When the last line in the proof is equal to the goal and the entire proof is shown to be true using checkProof	When the last line in the proof is not equal to the goal or the entire proof is shown to be false using checkProof	Checks if the Premises, Goal and Proof can form a valid proof within natural deduction
checkProof	When each line in the proof is true	When one line in the proof is false	Used for iterating over each line in the proof
checkProofLine: premise case	When the premise can be found within the given list of premises	When the premise can't be found within the given list of premises	Used for seeing if the current line in the proof is valid (for all cases of

			this predicate)
checkProofLine: assumption case	True as long as this line is not the last in the proof.		
checkProofLine: assumption based rule case	True if checkArgs and checkLevelsAssumption are passed as well as the specific function call (such as orel or negint)	False when it does not fulfil all predicates within the rule checkProofLine.	
checkProofLine: non assumption based rule case	True if checkArgs and checkLevelsAssumption are passed as well as the specific function call (such as orel or negint)	False when it does not fulfill all predicates within the rule checkProofLine.	
checkProofLine: lem	True is lem(Term) is fulfilled.	False when it does not fulfill all predicates within the rule checkProofLine.	

Table 2: All rules within natural deduction

Let X = a line number, then F(X) = a term found at line X in the proof. The first argument of F(X) is not X. For example F(X) = and(p,q) will have p,q as its 1st and 2nd arguments respectively.

Term is the formula at the line we are calling from.

The symbol "_" within the descriptions below means that any formula can be there unless specified explicitly. Term can be thought of as the formula below the line in a rule within natural deduction.

Predicate	The rule is true when:	The rule is false when:
andint(X,Y)	The term is of form and(_,_) and has formulas F(X) and F(Y) as arguments	The correct values of F(X) cannot be found within the proof (applies to all predicates in this table). This can occur when the rule points the wrong line or to a line that does not exist.

andel1(X)	F(X) is of form and(_,_) and has Term has the 1st argument	
andel2(X)	F(X) is of form and(_,_) and has Term has the 2nd argument	
orint1(X)	The first argument of Term is F(X)	
orint2(X)	The second argument of Term is F(X)	
orel(X,Y,U,V,W)	F(X)=or(F(Y),F(V)) and F(U)=F(V)=Term	
negnegint(X)	Term=neg(neg(F(X))	
negnegel(X)	F(X)=neg(neg(Term))	
impint(X,Y)	Term=imp(F(X),F(Y))	
impel(X,Y)	F(Y)=imp(F(X),Term)	
negel(X,Y)	Term=cont, F(Y)=neg(F(X))	
negint(X,Y)	F(Y)=cont, Term=neg(F(X))	
contel(X)	F(X)=cont	
copy(X)	F(X)=Term	
mt(X,Y)	Term=The negation of the 1st argument of F(X). F(Y) = The negation of the 2nd argument of F(X).	
lem	neg(1st argument of Term) = 2nd argument of Term	
pbc	F(X)=neg(Term). F(Y)=cont	
assumption	This always fails. Exists in case assumption is not treated properly in checkProofLine	

premise	This always fails. Exists in case premise is not treated properly in checkProofLine	

Table 3:

This table contains miscellaneous predicates that help the program with searching for formulas and verifying their properties.

Predicate	When this predicate is true	When this predicate is false	The purpose of this predicate.
find1stOfList find2ndOfList find3rdOfList	When the second argument of this predicate is the nth argument of the list.	When a list of the wrong size is used. find3rdOfList cannot be used on a list with a length of 1.	Used for finding the nth element of a list where n=1,2 or 3
findInProof	When Term is what is found by using a line number N and a Proof	When Term cannot be found within the proof.	finds a term within a proof when given a line number
searchInList	When the second argument of this predicate exists within a list (which is given as the 1st argument).	When the element we are looking for is not in the list.	Iterates over each line in a list until the current line is the one we are looking for.
isList	When the argument is a list	When the argument is not a list	For seeing if a variable is a list.
lineVerifPrint	Always true regardless of input		For debugging purposes, prints out when a line has been verified.
checkArgs	When all arguments are numbers and the line we are calling from is greater than the lines we are calling to.	When one of the arguments is greater than or equal to the line we are calling from.	For checking if the line we are calling from refers to previous lines in the proof only

isGre	When the first argument is greater than all of the other arguments.	When the first argument is less than or equal to	For seeing when a number is greater than several others.
levelInProof	When ResLevel=Level which occurs when we have found the correct level for line N in a Proof.	When the line N cannot be found within the proof.	Finds the level of a line within a proof.
checkLevels	When the current line and the arguments have the appropriate levels.	When the current level and arguments have the wrong levels.	Checks if the line we are calling from and the arguments have valid levels.
checkLevelsAssumption	When the current line and the arguments have the appropriate levels.	When the current level and arguments have the wrong levels.	same as checkLevels but for assumption rules.

Discussion

This section aims to answer the following question, can this proof verification program be utilised to generate proofs for sequents and how the handling of specific cases would occur.

It is in fact possible to make a program that can generate proofs for propositional logic sequents using natural deduction. The computer would start off with the premises and perform rules on them in order to get the desired result. It can do this by looking for specific patterns, similar to how our brains do when coming up with such proofs.

When dealing with sequents where the right hand side is of the form $A \rightarrow B$, we assume A and through several rules we can then deduce B. For sequents with a negation symbol in front of the right hand side term (for example $\neg(A \wedge (B \rightarrow C))$, we can assume the formula without the negation and show it to be false using negation introduction. These are examples of patterns that work well for starting a proof given that information about the right hand side.

Using the sequent $p,q \vdash p \wedge q$, here is an example of how the program could think when attempting to give a proof. The program knows that p and q are premises. What needs to be done here is getting p and q to be formed together into the right hand side. One possibility is that the program goes over each and every rule it knows and tests each until one of them produces the desired result. It then adds this information within a list, thus forming a proof.

Here is an example using the sequent, imp(p,q) ⊢imp(neg(q),and(neg(p),neg(q))). In this case, program sees the implication symbol in the goal so it will assume neg(q). It looks at the premises but it cannot use those directly. So it attempts to deconstruct imp(p,q) into smaller terms that it can use. It can do this using the rule Modus Tollens. This will result in neg(p) so now it knows this along with neg(q). Now it can use and introduction to form and(neg(p),neg(q)). Finally, it sees that the head and tail of this box are the arguments of the goal so it is done.

Such algorithms work by deconstructing the premises and reconstructing them to form the goal. So how can our existing code be used for creating a proof generating program ? One possible way is by using the defined natural deduction rules. The program can try a rule and see if it produces the desired result. If it does, it adds that rule as well its result to a new line within the proof. The program could search through all rules to see if any of them work or it could limit the search process by choosing rules that are related to a term. If we have the premise and(p,q) and we want to show neg(neg(p)), we would want it to use and elimination so on and(p,q) so we restrict it by only looking through and elimination and introduction rules. Once a proof has been generated, the verification program can be used to help us see if the proof generator algorithm generated a valid proof.

Appendix

```
[neg(neg(imp(p, neg(p))))].
p.

[
 [1, neg(neg(imp(p,neg(p)))), premise],
 [2, imp(p, neg(p)), negnegel(1)],
 [
  [3, p, assumption]
 ],
 [4, p, copy(3)]
].
```

An example of an invalid proof. This proof attempts to copy line 3 which lies within as assumption box. Another thing that is wrong with this proof is that nothing comes after the assumption line within the box.

```
[neg(neg(imp(p, neg(p))))].
neg(p).

[
 [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)]
].
```

This is an example of a valid proof .