Natural Deduction Proof Verifier

Konrad M. L. Claesson
konradcl@kth.se
Lab 2 – DD1351

November 23, 2019

1 Introduction

Natural deduction is a proof calculus comprised of a set of inference rules that are used to infer formulas from other formulas. The rules can be categorized into introduction and elimination rules. The former combine or negate formulas by introducing connectives; the latter decompose them by eliminating connectives. These rules are defined in terms of formula patterns that must hold true for a connective to be introduced or eliminated. For instance, to claim $\phi \wedge \psi$, it must be known that both of the formulas ϕ and ψ are true. Prolog is a logic programming language that enables easy expression of logical statements. To exemplify, the Prolog code

```
rule(_, [_, and(X, Y), andint(R1, R2)], Verified) :-
    member([R1, X, _], Verified),
    member([R2, Y, _], Verified).
```

states that $\phi \wedge \psi$ can be concluded, by the and introduction rule, if both ϕ and ψ are known to be true.

This report presents a Prolog-based algorithm that can verify the validity of any proof in propositional logical that has been written exclusively using the rules of natural deduction listed in the below table.

Prolog	Logisk	Prolog	Logisk
`premise`	premise	`impel(x,y)`	$\rightarrow$e x,y
`assumption`	assumption	`negint(x,y)`	$\neg$i x-y
`copy(x)`	copy x	`negel(x,y)`	$\neg$e x,y
`andint(x,y)`	$\wedge$i x, y	`contel(x)`	$\bot$e x
`andel1(x)`	$\wedge e_1$ x	`negnegint(x)`	$\neg\neg$i x
`andel2(x)`	$\wedge e_2$ x	`negnegel(x)`	$\neg\neg$e x
`orint1(x)`	$\vee i_1$ x	`mt(x,y)`	MT x,y
`orint2(x)`	$\vee i_2$ x	`pbc(x,y)`	PBC x-y
`orel(x,y,u,v,w)`	$\vee$e x,y–u,v–w	`lem`	LEM
`impint(x,y)`	$\rightarrow$i x–y		

Figure 1: Algorithm-Supported Proof Rules

2 Method

A proof is valid if all of its line are valid. A line is valid if it can be inferred by applying the rules of natural deduction to the proof's premises and previously inferred formulas. Accordingly, the validity of a proof can be determined if, and only if, the premises and previously derived formulas are known, and the required conditions for the employment of all used rules are defined. To this end, 19 instances of the `rule/3` predicate were defined, one for each rule from TABLE 1, that given the premises, a line, and all previous lines, returns `true` if the passed in line can be inferred from the premises and previous lines, and `false` otherwise. The set of all `rule/3` predicates achieve this behavior by, first, matching the rule used to deduce the passed in line (which is specified at the end of the line) with the predicate handling that rule; and secondly, verifying that the premises and derived formulas that were used to invoke the rule satisfy the conditions for its invocation. The premises of a proof are specified in the input file to the algorithm and stored in a list called `Premises`. Verified lines are appended to a list, usually named `Verified`, immediately after a `rule/3` predicate has declared them as `true`.

Naturally, handling of the algorithm-supported rules breaks down into premise handling, handling of inference rules, and assumption handling. How each are handled is described in sections 2.3.1, 2.3.2, and 2.3.3, respectively. Thereafter, two examples of how the algorithm verifies proofs are given.

2.1 Input Handling

Every input file is comprised of three Prolog terms:

1. A list of premises (left-hand side of sequent).
2. A conclusion (right-hand side of sequent).
3. A proof in natural deduction.

For example, the sequent $\neg(p \vee q) \vdash \neg p \wedge \neg q$ with the proof

```
INSERT PROOF
```

is entered into the algorithm in the below format.

```
INSERT PROLOG-FORMATED PROOF
```

In this format, the premises, conclusion and proof can be read into `Premises`, `Conclusion` and `Proof`, respectively, and fed into the proof verification algorithm `verify_proof/3`, via the following code.

```
verify(InputFileName) :-
  see(InputFileName),
  read(Premises), read(Conclusion), read(Proof),
  seen,
  valid_proof(Premises, Conclusion, Proof).
```

2.2 The Recursive Helper Predicates

When the input file has been read, the verification algorithm is initiated by a call to `valid_proof/3`.

```
VALID PROOF CODE
```

This helper predicate takes in the `Premises`, `Proof` and `Conclusion` from the input file and invokes the `verify_end(Conclusion, Proof)` and `verify_proof(Premises, Proof, [])` predicates.

`verify_end(Conclusion, Proof)` ensures that the final line of the proof is equal to the sequent's conclusion.

```
VERIFY END CODE
```

This is achieved by first applying Prolog's built-in `last/3` predicate to extract the last element (line) in `Proof`, and then utilizing `nth0/3` to select the formula present on the last line. If the proof is valid, this formula is equal to the sequent's conclusion. Accordingly, `verify_end` concludes with an equality check that ensures that the described equality holds.

`verify_proof(Premises, Proof, Verified)` recursively validates the provided proof line-by-line, from top to bottom.

```
VERIFY PROOF CODE
```

It validates each line (element in `Proof`) by invoking `rule/3` with the premises, line, and previously verified lines as arguments. `rule/3` then returns `true` whenever the line follows by natural deduction from the premises and previously verified lines, and false otherwise. If a line is valid, it gets appended to the `Verified` list and `verify_proof` is called again with the yet unverified lines and the new list of verified lines.

2.3 The Rule Predicate

`rule/3` takes in the list of premises, a line or assumption box, and the, as of then, verified lines of a proof. It then ensures that the line or assumption box follows by natural deduction from the premises and previous lines of the proof. It does this by inspecting the last element of each line, which specifies the proof rule that was applied to deduce the formula on that line, and then checking that the conditions for using the rule are fulfilled.

In the subsections ahead the various cases of `rule/3` are explored in greater detail.

2.3.1 Premise Handling

Lines that claim to be premises are verified by checking that the `Formula` on the given line is a member of the proof's `Premises`. This is done by the below code,

```
rule(Premises, [_, Formula, premise], _) :-
    member(Formula, Premises).
```

where the `member/2` predicate resolves whether `Formula` is a member of `Premiss`.

2.3.2 Handling of Inference Rules

As described earlier, a line gets matched with a `rule/3` predicate based on the inference rule that was used to derive it. The following states under what conditions each of the 17 inference rules evaluate to `true`. An implementation of each rule can be found in the algorithm source code in the appendix.

INSERT TABLE

2.3.3 Assumption Handling

The most outstanding `rule/3` predicate is the one that handles assumptions. There are two reasons for this. Firstly, rather than being matched with lines that declare themselves as "derived by assumption", it is matched to boxes (implemented as a list of lines) that start with an assumption. Secondly, unlike the other predicates, it does not check for the fulfillment of a set of predefined conditions; instead, it treats the lines of the box as a proof in its own right, where the assumption becomes a premise, and returns `true` only if the proof within the box is valid. In this way, the algorithm handles assumptions by regarding their associated boxes as sub-proofs that can be verified recursively.

The subsequent code handles assumptions in the described manner,

```
rule(Premises, [[R, Formula, assumption] | T], Verified) :-
```

```
append(Verified, [[R, Formula, assumption]], VerifiedNew),
verify_proof(Premises, T, VerifiedNew).
```

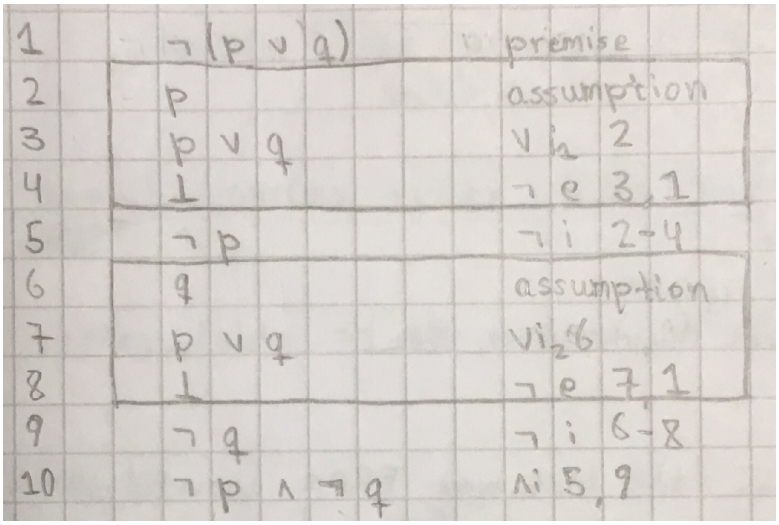
where `Premises` is the list of premises supplied in the input file, `[[R, Formula, assumption] | T]` is the assumption box, and `Verified` is the list of verified lines. The `append/3` predicate appends the assumption line to the list of verified lines. Then, `verify_proof` is recursively called to verify the sub-proof within the box.

2.4 Example of Proof Verification

In this subsection, two examples of how the proof verification algorithm works are provided. First, a detailed explanation of how the algorithm progresses from accepting inputs to concluding the validity of a coherent proof is given. Then, a shorter description of the algorithm's behavior in the event of an invalid proof is presented.

2.4.1 Verification of a Valid Proof

The following is a valid proof of one of de Morgan's laws. Specifically, it proves the sequent $\neg(p \vee q) \vdash \neg p \wedge \neg q$.



In Prolog notation, this proof is written:

```
[1,      neg(or(p, q)),          premise],
```

```
[
   [2,   p,                      assumption],
   [3,   or(p, q),               orint1(2)],
   [4,   cont,                   negel(3, 1)]
],
[5,      neg(p),                negint(2, 4)],
[
   [6,   q,                      assumption],
   [7,   or(p, q),               orint2(6)],
   [8,   cont,                   negel(7, 1)]
],
[9,      neg(q),                negint(6, 8)],
[10,     and(neg(p), neg(q)),   andint(5, 9)]
```

The proof verification algorithm (view appendix) successfully identifies this as a valid proof. To do this, the algorithm initiates by calling `verify(InputFileName)`, which reads in the proof from the specified file (see Section 2.1). Then, `valid_proof(Premises, Conclusion, Proof)` is invoked, which in turn calls `verify_end(Conclusion, Proof)` and `verify_proof(Premises, Proof, [])`, in the given order (see Section 2.2). The `verify_proof/3` predicate recursively verifies each line by invoking `rule/3` and calling itself. In the above proof, the first line invokes the premise verification rule

```
rule(Premises, [_, Formula, premise], _) :-
   member(Formula, Premises).
```

which checks that `Formula` (the formula on the given line) is a member of `Premises`. If `Formula` is a member, the line is valid and `rule` returns `true`. Line two is an assumption and thus opens a box. It is handled by the assumption rule

```
rule(Premises, [[R, Formula, assumption] | T], Verified) :-
   append(Verified, [[R, Formula, assumption]], VerifiedNew),
   verify_proof(Premises, T, VerifiedNew).
```

which returns `true` if the sub-proof of the assumption box is valid. To verify that the assumption box contains a valid proof the assumption line is appended to the list of verified lines, and `verify_proof/3` is called to validate the sub-proof. Within the box, line three is verified by ensuring that its application of the first disjunction introduction rule is valid. The below code does this

```
rule(_, [_, or(X, _), orint1(R)], Verified) :-
   member([R, X, _], Verified).
```

by confirming that the formula p has been deduced earlier in the proof. Line four is deduced by administering the rule of negation elimination. The below code verifies that the rule can be employed

```
rule(_, [_, cont, negel(R1, R2)], Verified) :-
   member([R1, X, _], Verified),
   member([R2, neg(X), _], Verified).
```

by checking that both $p \vee q$ and $\neg(p \vee q)$ occur previously in the proof (are members of `Verified`). When the sub-proof of the assumption box has been validated, the entire box is appended to `Verified` before continuing to verify the next line or assumption box. This ensures that no single line that is predicated on the assumption can be referenced without also referencing the assumption. In the above proof, the fifth line, which also is the first line following the upper assumption box, is deduced by applying the rule of negation introduction. The subsequent code verifies that the rule can be applied

```
rule(_, [_, neg(X), negint(R1, R2)], Verified) :-
   member([[R1, X, assumption] | T], Verified),
   last(T, BoxConclusion),
   [R2, cont, _] = BoxConclusion.
```

by establishing that the assumption box starts with p, and that it ends with a contradiction.

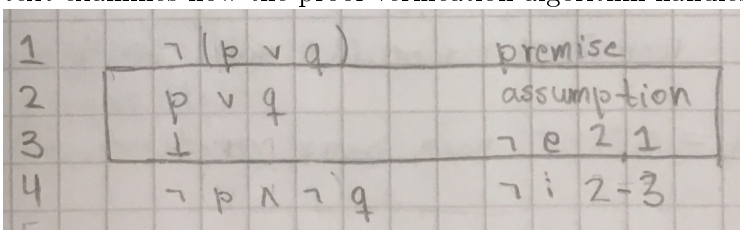
Lines six through nine repeat the same deductive pattern as lines two through five. The concluding line of the proof is arrived at by utilizing the rule of conjunction introduction. The employment of this rule is validated by the below code

```
rule(_, [_, and(X, Y), andint(R1, R2)], Verified) :-
    member([R1, X, _], Verified),
    member([R2, Y, _], Verified).
```

which states that the rule can be administered if, and only if, both $\neg p$ and $\neg q$ occur previously in the proof (exist in `Verified`).

2.4.2 Verification of an Invalid Proof

The following is an invalid proof of the same sequent as above. Lines one through three are deduced by rules that have already been addressed, and hence they will not be discussed any further. Nevertheless, the fourth and final line incorrectly applies the rule of negation introduction. The subsequent text examines how the proof verification algorithm handles this.



Recall that the negation introduction rule can be employed to conclude $\neg\phi$, for some formula ϕ, if, and only if, there exists a box starts with an assumption of ϕ and ends in a contradiction. The proof passes the latter requirement, which is enforced by the code

```
last(T, BoxConclusion),
[R2, cont, _] = BoxConclusion.
```

but fails the former, as the assumption $\phi = p \vee q$ combined with the contradiction and negation introduction rule, can only be used to conclude $\neg(p \vee q)$, and not $\neg p \vee \neg q$. Hence, the called `rule/3` predicate fails. This failure propagates through `verify_proof/3` and `valid_proof/3` all the way to `verify/1`, where it ultimately causes the proof to be declared as invalid.

3 Results

The algorithm correctly identified the validity of every minimal, rule-specific, proof. It also accurately verified each of the XX multi-step proofs.

The algorithm along with the proofs in an input-ready format can be found by following this link: `github.com/konradcl/prolog-proof-verifier`.

Predicate	**Parameters**	**Truth Conditions**
`verify`	`InputFileName`	`valid_proof`

`valid_proof`	`Premise`
	`Conclusion`
	`Proof`
		`verify_end`
		`verify_proof`

`verify_end`	`Conclusion` `Proof`	Proof ends with sequent's conclusion.

`verify_proof`	`Premises`
	`Proof`
	`Verified`
		The first line, and recursively all subsequent lines, are provable by natural deduction.

`rule`	`Premises`
	`Line/Box`
	`Verified`
		If the second argument is an assumption box, `rule` is true if the sub-proof of the box is valid (i.e. if `verify_proof` returns true when invoked with the sub-proof as its `Proof` argument).
		
		If the second argument is a line, the conditions determining the return value of `rule` differ based on what deduction rule was used to arrive at the formula on the line.
		
		For lines that are *premises*, the `Formula` on the line must exist in the `Premises` of the proof.
		
		For lines that are deduced by the *copy* rule, the copied formula must appear previously in the proof (i.e. be a member of `Verified`).
		
		For lines with a formula $\neg\neg\phi$, that is deduced by *double negation, introduction*, the formula ϕ must appear previously in the proof (i.e. exist in `Verified`).
		
		For lines with a formula ϕ that is deduced by *double negation elimination*, the formula $\neg\neg\phi$ must appear previously in the proof (i.e exist in `Verified`).

Predicate	Parameters	Truth Conditions
		For a line with a formula $\phi \wedge \psi$ that is deduced by *and introduction*, the formulas ϕ and ψ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula ϕ, that is deduced by *first and elimination*, a formula of the form $\phi \wedge \psi$ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula ϕ, that is deduced by *second and elimination*, a formula of the form $\psi \wedge \phi$ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula $\phi \vee \psi$, that is deduced by *first or introduction*, a formula ϕ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula $\phi \vee \psi$, that is deduced by *second or introduction*, a formula ψ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula χ, that is deduced by *or elimination*, a formula $\phi \vee \psi$, a box assuming ϕ and ending in χ, and a box assuming ψ and ending in χ, must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula $\phi \rightarrow \psi$, that is deduced by *implication introduction*, a box assuming ϕ and concluding in ψ must appear previously in the proof (i.e. exist in `Verified`)

Predicate	Parameters	Truth Conditions
		For a line with a formula ψ that is deduced by *implication elimination* a formula ϕ, and a box assuming ϕ and ending in ψ, must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula $\neg\phi$ that is deduced by *negation introduction* a box assuming ϕ and concluding in $\bot$ must appear previously in the proof (i.e. exist in `Verified`). For a line with a contradiction $\bot$ that is deduced by *negation elimination* two formulas ϕ and $\neg\phi$ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula ϕ that is deduced by *contradiction elimination* a line with a contradiction $\bot$ must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula $\neg\phi$ that is deduced by *modus tollens* a formula $\phi \to \psi$, and a formula $\neg\psi$, must appear previously in the proof (i.e. exist in `Verified`). For a line with a formula ϕ that is derived by a *proof by contradiction* a box assuming $\neg\phi$ and ending in a contradiction $\bot$ must appear previously in the proof (i.e. exist in `Verified`). By the *law of the excluded middle* a line with a formula of the form $\phi \vee \neg\phi$ is always valid.